

Eighth Edition

JAVA™ PROGRAMMING

Joyce Farrell

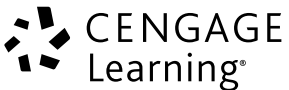


JAVA[™] PROGRAMMING

EIGHTH EDITION

JAVA™ PROGRAMMING

JOYCE FARRELL



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

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please visit www.cengage.com/highered to search by ISBN#, author, title, or keyword for materials in your areas of interest.

Important Notice: Media content referenced within the product description or the product text may not be available in the eBook version.

Java Programming,
Eighth Edition
Joyce Farrell

Product Director:
Kathleen McMahon

Senior Content Developer:
Alyssa Pratt

Development Editor: Dan Seiter

Marketing Manager: Eric LaScola

Manufacturing Planner:
Julio Esperas

Art Director: Jack Pendleton

**Production Management,
Copyediting, Composition,
Proofreading, and Indexing:**
Integra Software Services Pvt. Ltd.

Cover Photo:
©Maram/Shutterstock.com

© 2016, 2014, 2012 Cengage Learning

WCN: 02-200-203

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored, or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706

For permission to use material from this text or product,
submit all requests online at www.cengage.com/permissions.

Further permissions questions can be emailed to
permissionrequest@cengage.com.

Library of Congress Control Number: 2014956152

ISBN: 978-1-285-85691-9

Cengage Learning
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at www.cengage.com/global.

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

All images © 2016 Cengage Learning®. All rights reserved.

To learn more about Cengage Learning Solutions, visit
www.cengage.com.

Purchase any of our products at your local college store
or at our preferred online store www.cengagebrain.com.

Brief Contents

	Preface	xxi
CHAPTER 1	Creating Java Programs	1
CHAPTER 2	Using Data	53
CHAPTER 3	Using Methods, Classes, and Objects	119
CHAPTER 4	More Object Concepts	183
CHAPTER 5	Making Decisions	245
CHAPTER 6	Looping	301
CHAPTER 7	Characters, Strings, and the <code>StringBuilder</code>	353
CHAPTER 8	Arrays	393
CHAPTER 9	Advanced Array Concepts	439
CHAPTER 10	Introduction to Inheritance	491
CHAPTER 11	Advanced Inheritance Concepts	537
CHAPTER 12	Exception Handling	593
CHAPTER 13	File Input and Output	665
CHAPTER 14	Introduction to Swing Components	729
CHAPTER 15	Advanced GUI Topics	791
CHAPTER 16	Graphics	861
APPENDIX A	Working with the Java Platform	919
APPENDIX B	Data Representation	925
APPENDIX C	Formatting Output	931
APPENDIX D	Generating Random Numbers	941
APPENDIX E	Javadoc	949
	Glossary	957
	Index	979

Contents

	Preface	xxi
CHAPTER 1	Creating Java Programs	1
	Learning Programming Terminology	2
	Comparing Procedural and Object-Oriented Programming Concepts	6
	Procedural Programming	6
	Object-Oriented Programming	6
	Understanding Classes, Objects, and Encapsulation	7
	Understanding Inheritance and Polymorphism	9
	Features of the Java Programming Language	11
	Java Program Types	12
	Analyzing a Java Application that Produces Console Output	13
	Understanding the Statement that Produces the Output	14
	Understanding the First Class	15
	Indent Style	18
	Understanding the main() Method	19
	Saving a Java Class	21
	Compiling a Java Class and Correcting Syntax Errors	23
	Compiling a Java Class	23
	Correcting Syntax Errors	24
	Running a Java Application and Correcting Logic Errors	29
	Running a Java Application	29
	Modifying a Compiled Java Class	30
	Correcting Logic Errors	31
	Adding Comments to a Java Class	32
	Creating a Java Application that Produces GUI Output	35
	Finding Help	38
	Don't Do It	39
	Key Terms	41

Chapter Summary 45
Review Questions 46
Exercises 48
 Programming Exercises 48
 Debugging Exercises 50
 Game Zone 50
 Case Problems 51

CHAPTER 2

Using Data 53
 Declaring and Using Constants and Variables 54
 Declaring Variables 55
 Declaring Named Constants 56
 The Scope of Variables and Constants 58
 Concatenating Strings to Variables and Constants 58
 Pitfall: Forgetting that a Variable Holds
 One Value at a Time 60
 Learning About Integer Data Types 64
 Using the boolean Data Type 70
 Learning About Floating-Point Data Types 71
 Using the char Data Type 72
 Using the Scanner Class to Accept Keyboard Input 78
 Pitfall: Using nextLine() Following One of the
 Other Scanner Input Methods 81
 Using the JOptionPane Class to Accept GUI Input 87
 Using Input Dialog Boxes 87
 Using Confirm Dialog Boxes 91
 Performing Arithmetic 93
 Associativity and Precedence 95
 Writing Arithmetic Statements Efficiently 96
 Pitfall: Not Understanding Imprecision
 in Floating-Point Numbers 96
 Understanding Type Conversion 101
 Automatic Type Conversion 101
 Explicit Type Conversions 102
 Don't Do It 106
 Key Terms 107

Chapter Summary	111
Review Questions	111
Exercises	114
Programming Exercises	114
Debugging Exercises	116
Game Zone	117
Case Problems	118
CHAPTER 3 Using Methods, Classes, and Objects	119
Understanding Method Calls and Placement	120
Understanding Method Construction	123
Access Specifiers	123
Return Type	124
Method Name	125
Parentheses	125
Adding Parameters to Methods	129
Creating a Method that Receives a Single Parameter	130
Creating a Method that Requires Multiple Parameters	133
Creating Methods that Return Values	136
Chaining Method Calls	138
Learning About Classes and Objects	142
Creating a Class	145
Creating Instance Methods in a Class	147
Organizing Classes	150
Declaring Objects and Using their Methods	154
Understanding Data Hiding	156
An Introduction to Using Constructors	159
Understanding that Classes Are Data Types	163
Don't Do It	168
Key Terms	168
Chapter Summary	170
Review Questions	171
Exercises	174
Programming Exercises	174
Debugging Exercises	177
Game Zone	178
Case Problems	179

CHAPTER 4 **More Object Concepts 183**

- Understanding Blocks and Scope 184
- Overloading a Method 192
 - Automatic Type Promotion in Method Calls 194
- Learning About Ambiguity 199
- Creating and Calling Constructors with Parameters 200
 - Overloading Constructors 201
- Learning About the `this` Reference 205
 - Using the `this` Reference to Make Overloaded Constructors
More Efficient 209
- Using `static` Fields 213
 - Using Constant Fields 215
- Using Automatically Imported, Prewritten Constants
and Methods 220
 - The `Math` Class 221
 - Importing Classes that Are Not Imported Automatically 223
 - Using the `LocalDate` Class 224
- Understanding Composition and Nested Classes 230
 - Composition 230
 - Nested Classes 232
- Don't Do It 234
- Key Terms 234
- Chapter Summary 236
- Review Questions 236
- Exercises 239
 - Programming Exercises 239
 - Debugging Exercises 242
 - Game Zone 242
 - Case Problems 243

CHAPTER 5 **Making Decisions 245**

- Planning Decision-Making Logic 246
- The `if` and `if...else` Statements 248
 - The `if` Statement 248
 - Pitfall: Misplacing a Semicolon in an `if` Statement 249
 - Pitfall: Using the Assignment Operator Instead
of the Equivalency Operator 250

Pitfall: Attempting to Compare Objects	
Using the Relational Operators	251
The <code>if...else</code> Statement	251
Using Multiple Statements in <code>if</code> and <code>if...else</code> Clauses	254
Nesting <code>if</code> and <code>if...else</code> Statements	260
Using Logical AND and OR Operators	263
The AND Operator	263
The OR Operator	265
Short-Circuit Evaluation	266
Making Accurate and Efficient Decisions	269
Making Accurate Range Checks	270
Making Efficient Range Checks	272
Using <code>&&</code> and <code> </code> Appropriately	273
Using the <code>switch</code> Statement	274
Using the Conditional and NOT Operators	280
Using the NOT Operator	281
Understanding Operator Precedence	282
Adding Decisions and Constructors	
to Instance Methods	285
Don't Do It	289
Key Terms	289
Chapter Summary	291
Review Questions	291
Exercises	294
Programming Exercises	294
Debugging Exercises	297
Game Zone	297
Case Problems	299

CHAPTER 6	Looping	301
	Learning About the Loop Structure	302
	Creating <code>while</code> Loops	303
	Writing a Definite <code>while</code> Loop	303
	Pitfall: Failing to Alter the Loop Control Variable	
	Within the Loop Body	305
	Pitfall: Unintentionally Creating a Loop with	
	an Empty Body	306

- Altering a Definite Loop's Control Variable 307
- Writing an Indefinite while Loop 308
- Validating Data 310
- Using Shortcut Arithmetic Operators 314
- Creating a for Loop 319
 - Unconventional for Loops 320
- Learning How and When to Use a do...while Loop 325
- Learning About Nested Loops 328
- Improving Loop Performance 333
 - Avoiding Unnecessary Operations 333
 - Considering the Order of Evaluation of Short-Circuit Operators 334
 - Comparing to Zero 334
 - Employing Loop Fusion 336
 - Using Prefix Incrementing Rather than Postfix Incrementing 337
 - A Final Note on Improving Loop Performance 338
- Don't Do It 342
- Key Terms 342
- Chapter Summary 344
- Review Questions 344
- Exercises 347
 - Programming Exercises 347
 - Debugging Exercises 350
 - Game Zone 350
 - Case Problems 352

CHAPTER 7

- Characters, Strings, and the StringBuilder 353**
- Understanding String Data Problems 354
- Using Character Class Methods 355
- Declaring and Comparing String Objects 359
 - Comparing String Values 359
 - Empty and null Strings 363
- Using Other String Methods 365
 - Converting String Objects to Numbers 369

Learning About the <code>StringBuilder</code> and <code>StringBuffer</code> Classes	374
Don't Do It	381
Key Terms	382
Chapter Summary	382
Review Questions	383
Exercises	385
Programming Exercises	385
Debugging Exercises	388
Game Zone	388
Case Problems	391
CHAPTER 8 Arrays	393
Declaring Arrays	394
Initializing an Array	399
Using Variable Subscripts with an Array	402
Using the Enhanced <code>for</code> Loop	403
Using Part of an Array	404
Declaring and Using Arrays of Objects	406
Using the Enhanced <code>for</code> Loop with Objects	408
Manipulating Arrays of <code>Strings</code>	408
Searching an Array and Using Parallel Arrays	414
Using Parallel Arrays	415
Searching an Array for a Range Match	418
Passing Arrays to and Returning Arrays from Methods	422
Returning an Array from a Method	426
Don't Do It	428
Key Terms	428
Chapter Summary	429
Review Questions	430
Exercises	433
Programming Exercises	433
Debugging Exercises	435
Game Zone	435
Case Problems	438

CHAPTER 9	Advanced Array Concepts	439
	Sorting Array Elements Using the Bubble Sort Algorithm	440
	Using the Bubble Sort Algorithm	440
	Improving Bubble Sort Efficiency	442
	Sorting Arrays of Objects	443
	Sorting Array Elements Using the Insertion Sort Algorithm	448
	Using Two-Dimensional and Other Multidimensional Arrays	452
	Passing a Two-Dimensional Array to a Method	454
	Using the Length Field with a Two-Dimensional Array	455
	Understanding Ragged Arrays	456
	Using Other Multidimensional Arrays	456
	Using the Arrays Class	459
	Using the ArrayList Class	467
	Creating Enumerations	472
	Don't Do It	479
	Key Terms	479
	Chapter Summary	480
	Review Questions	481
	Exercises	484
	Programming Exercises	484
	Debugging Exercises	486
	Game Zone	487
	Case Problems	490
CHAPTER 10	Introduction to Inheritance	491
	Learning About the Concept of Inheritance	492
	Diagramming Inheritance Using the UML	492
	Inheritance Terminology	495
	Extending Classes	496
	Overriding Superclass Methods	502
	Using the @Override Tag	504
	Calling Constructors During Inheritance	507
	Using Superclass Constructors that Require Arguments	508
	Accessing Superclass Methods	513
	Comparing this and super	515
	Employing Information Hiding	516

Methods You Cannot Override	518
A Subclass Cannot Override <code>static</code> Methods in Its Superclass	518
A Subclass Cannot Override <code>final</code> Methods in Its Superclass	522
A Subclass Cannot Override Methods in a <code>final</code> Superclass	523
Don't Do It	525
Key Terms	525
Chapter Summary	526
Review Questions	527
Exercises	530
Programming Exercises	530
Debugging Exercises	533
Game Zone	534
Case Problems	535
CHAPTER 11 Advanced Inheritance Concepts	537
Creating and Using Abstract Classes	538
Using Dynamic Method Binding	547
Using a Superclass as a Method Parameter Type	549
Creating Arrays of Subclass Objects	551
Using the <code>Object</code> Class and Its Methods	554
Using the <code>toString()</code> Method	556
Using the <code>equals()</code> Method	559
Using Inheritance to Achieve Good Software Design	564
Creating and Using Interfaces	565
Creating Interfaces to Store Related Constants	570
Creating and Using Packages	574
Don't Do It	580
Key Terms	580
Chapter Summary	581
Review Questions	582
Exercises	585
Programming Exercises	585
Debugging Exercises	589
Game Zone	590
Case Problems	590

CHAPTER 12 **Exception Handling 593**

- Learning About Exceptions 594
- Trying Code and Catching Exceptions 599
 - Using a try Block to Make Programs “Foolproof” 604
 - Declaring and Initializing Variables in try...catch Blocks 606
- Throwing and Catching Multiple Exceptions 609
- Using the finally Block 615
- Understanding the Advantages of Exception Handling 618
- Specifying the Exceptions that a Method Can Throw 621
- Tracing Exceptions Through the Call Stack 626
- Creating Your Own Exception Classes 630
- Using Assertions 634
- Displaying the Virtual Keyboard 650
- Don't Do It 653
- Key Terms 654
- Chapter Summary 655
- Review Questions 656
- Exercises 659
 - Programming Exercises 659
 - Debugging Exercises 662
 - Game Zone 662
 - Case Problems 663

CHAPTER 13 **File Input and Output 665**

- Understanding Computer Files 666
- Using the Path and Files Classes 667
 - Creating a Path 668
 - Retrieving Information About a Path 669
 - Converting a Relative Path to an Absolute One 670
 - Checking File Accessibility 671
 - Deleting a Path 673
 - Determining File Attributes 674
- File Organization, Streams, and Buffers 678
- Using Java's IO Classes 680
 - Writing to a File 683
 - Reading from a File 685

Creating and Using Sequential Data Files	687
Learning About Random Access Files	693
Writing Records to a Random Access Data File	697
Reading Records from a Random Access Data File	704
Accessing a Random Access File Sequentially	704
Accessing a Random Access File Randomly	705
Don't Do It	719
Key Terms	719
Chapter Summary	720
Review Questions	721
Exercises	724
Programming Exercises	724
Debugging Exercises	726
Game Zone	727
Case Problems	727
CHAPTER 14 Introduction to Swing Components	729
Understanding Swing Components	730
Using the JFrame Class	731
Customizing a JFrame's Appearance	734
Using the JLabel Class	738
Changing a JLabel's Font	740
Using a Layout Manager	743
Extending the JFrame Class	746
Adding JTextFields, JButtons, and Tool Tips to a JFrame	748
Adding JTextFields	748
Adding JButtons	750
Using Tool Tips	752
Learning About Event-Driven Programming	755
Preparing Your Class to Accept Event Messages	756
Telling Your Class to Expect Events to Happen	757
Telling Your Class How to Respond to Events	757
An Event-Driven Program	757
Using Multiple Event Sources	759
Using the setEnabled() Method	761
Understanding Swing Event Listeners	764

Using the JCheckBox, ButtonGroup, and JComboBox	
Classes	767
The JCheckBox Class	767
The ButtonGroup Class	771
The JComboBox Class	772
Don't Do It	780
Key Terms	780
Chapter Summary	781
Review Questions	783
Exercises	785
Programming Exercises	785
Debugging Exercises	787
Game Zone	787
Case Problems	788
CHAPTER 15	Advanced GUI Topics 791
Understanding the Content Pane	792
Using Color	795
Learning More About Layout Managers	797
Using BorderLayout	798
Using FlowLayout	800
Using GridLayout	802
Using CardLayout	803
Using Advanced Layout Managers	805
Using the JPanel Class	813
Creating JScrollPane	821
A Closer Look at Events and Event Handling	824
An Event-Handling Example: ActionListener	827
Using AWTEvent Class Methods	830
Understanding x- and y-Coordinates	832
Handling Mouse Events	832
Using Menus	837
Using Specialized Menu Items	841
Using addSeparator()	843
Using setMnemonic()	843
Don't Do It	848
Key Terms	849

Chapter Summary	850
Review Questions	851
Exercises	853
Programming Exercises	853
Debugging Exercises	855
Game Zone	855
Case Problems	859

CHAPTER 16 Graphics 861

Learning About Rendering Methods	862
Drawing Strings	865
Repainting	867
Setting a Font	869
Using Color	870
Drawing Lines and Shapes	874
Drawing Lines	874
Drawing Unfilled and Filled Rectangles	875
Drawing Clear Rectangles	875
Drawing Rounded Rectangles	876
Drawing Shadowed Rectangles	878
Drawing Ovals	879
Drawing Arcs	880
Creating Polygons	881
Copying an Area	883
Using the <code>paint()</code> Method with <code>JFrames</code>	883
Learning More About Fonts	891
Discovering Screen Statistics	893
Discovering Font Statistics	894
Drawing with Java 2D Graphics	898
Specifying the Rendering Attributes	899
Setting a Drawing Stroke	901
Creating Objects to Draw	902
Don't Do It	910
Key Terms	911
Chapter Summary	911
Review Questions	912
Exercises	915

	Programming Exercises	915
	Debugging Exercises	916
	Game Zone	916
	Case Problems	918
APPENDIX A	Working with the Java Platform	919
	Learning about the Java SE Development Kit	920
	Configuring Windows to Use the JDK	920
	Finding the Command Prompt	921
	Command Prompt Anatomy	921
	Changing Directories	921
	Setting the <code>class</code> and <code>classpath</code> Variables	922
	Changing a File's Name	922
	Compiling and Executing a Java Program	923
	Key Terms	923
APPENDIX B	Data Representation	925
	Understanding Numbering Systems	926
	Representing Numeric Values	927
	Representing Character Values	929
	Key Terms	930
APPENDIX C	Formatting Output	931
	Rounding Numbers	932
	Using the <code>printf()</code> Method	933
	Specifying a Number of Decimal Places to Display with <code>printf()</code>	936
	Specifying a Field Size with <code>printf()</code>	937
	Using the Optional Argument Index with <code>printf()</code>	938
	Using the <code>DecimalFormat</code> Class	939
	Key Terms	940
APPENDIX D	Generating Random Numbers	941
	Understanding Computer-Generated Random Numbers	942
	Using the <code>Math.random()</code> Method	943
	Using the <code>Random</code> Class	944
	Key Terms	947

APPENDIX E

Javadoc	949
The Javadoc Documentation Generator	950
Javadoc Comment Types	950
Generating Javadoc Documentation	952
Specifying Visibility of Javadoc Documentation	955
Key Terms	956
Glossary	957
Index	979

Preface

Java Programming, Eighth Edition, provides the beginning programmer with a guide to developing applications using the Java programming language. Java is popular among professional programmers because it can be used to build visually interesting graphical user interface (GUI) and Web-based applications. Java also provides an excellent environment for the beginning programmer—a student can quickly build useful programs while learning the basics of structured and object-oriented programming techniques.

This textbook assumes that you have little or no programming experience. It provides a solid background in good object-oriented programming techniques and introduces terminology using clear, familiar language. The programming examples are business examples; they do not assume a mathematical background beyond high-school business math. In addition, the examples illustrate only one or two major points; they do not contain so many features that you become lost following irrelevant and extraneous details. Complete, working programs appear frequently in each chapter; these examples help students make the transition from the theoretical to the practical. The code presented in each chapter can also be downloaded from the publisher's Web site, so students can easily run the programs and experiment with changes to them.

The student using *Java Programming, Eighth Edition*, builds applications from the bottom up rather than starting with existing objects. This facilitates a deeper understanding of the concepts used in object-oriented programming and engenders appreciation for the existing objects students use as their knowledge of the language advances. When students complete this book, they will know how to modify and create simple Java programs, and they will have the tools to create more complex examples. They also will have a fundamental knowledge of object-oriented programming, which will serve them well in advanced Java courses or in studying other object-oriented languages such as C++, C#, and Visual Basic.

Organization and Coverage

Java Programming, Eighth Edition, presents Java programming concepts, enforcing good style, logical thinking, and the object-oriented paradigm. Objects are covered right from the beginning, earlier than in many other textbooks. You create your first Java program in Chapter 1. Chapters 2, 3, and 4 increase your understanding of how data, classes, objects, and methods interact in an object-oriented environment.

Chapters 5 and 6 explore input and repetition structures, which are the backbone of programming logic and essential to creating useful programs in any language. You learn the special considerations of string and array manipulation in Chapters 7, 8, and 9.

Chapters 10, 11, and 12 thoroughly cover inheritance and exception handling. Inheritance is the object-oriented concept that allows you to develop new objects quickly by adapting the features of existing objects; exception handling is the object-oriented approach to handling errors. Both are important concepts in object-oriented design. Chapter 13 provides information on handling files so you can permanently store and retrieve program output.

Chapters 14, 15, and 16 introduce GUI Swing components (Java's visually pleasing, user-friendly widgets), their layout managers, and graphics.

Features

The following features are new for the Eighth Edition:

- **JAVA 8E:** All programs have been tested using Java 8e, the newest edition of Java.
- **WINDOWS 8.1:** All programs have been tested in Windows 8.1, and all screen shots have been taken in this new environment.
- **DATE AND TIME CLASSES:** This edition provides thorough coverage of the `java.time` package, which is new in Java 8e.
- **ON-SCREEN KEYBOARD:** This edition provides instructions for displaying and using an on-screen keyboard with either a touch screen or a standard screen.
- **MODERNIZED GRAPHICS OUTPUT:** The chapter on graphics (Chapter 16) has been completely rewritten to focus on Swing component graphics production using the `paintComponent()` method.
- **MODERNIZED OVERRIDING:** The `@Override` tag is introduced.
- **EXPANDED COVERAGE OF THE `EQUALS()` METHOD:** The book provides a thorough explanation of the difference between overloading and overriding the `equals()` method.
- **PROGRAMMING EXERCISES:** Each chapter contains several new programming exercises not seen in previous editions. All exercises and their solutions from the previous edition that were replaced in this edition are still available in the Instructor's Resource Kit.

Additionally, *Java Programming, Eighth Edition*, includes the following features:

- **OBJECTIVES:** Each chapter begins with a list of objectives so you know the topics that will be presented in the chapter. In addition to providing a quick reference to topics covered, this feature provides a useful study aid.
- **YOU DO IT:** In each chapter, step-by-step exercises help students create multiple working programs that emphasize the logic a programmer uses in choosing statements to include. These sections provide a means for students to achieve success on their own—even those in online or distance learning classes.
- **NOTES:** These highlighted tips provide additional information—for example, an alternative method of performing a procedure, another term for a concept, background information on a technique, or a common error to avoid.

- **EMPHASIS ON STUDENT RESEARCH:** The student frequently is directed to the Java Web site to investigate classes and methods. Computer languages evolve, and programming professionals must understand how to find the latest language improvements. This book encourages independent research.
- **FIGURES:** Each chapter contains many figures. Code figures are most frequently 25 lines or fewer, illustrating one concept at a time. Frequent screen shots show exactly how program output appears. Callouts appear where needed to emphasize a point.
- **COLOR:** The code figures in each chapter contain all Java keywords in blue. This helps students identify keywords more easily, distinguishing them from programmer-selected names.
- **FILES:** More than 200 student files can be downloaded from the publisher’s Web site. Most files contain the code presented in the figures in each chapter; students can run the code for themselves, view the output, and make changes to the code to observe the effects. Other files include debugging exercises that help students improve their programming skills.
- **TWO TRUTHS & A LIE:** A short quiz reviews each chapter section, with answers provided. This quiz contains three statements based on the preceding section of text—two statements are true and one is false. Over the years, students have requested answers to problems, but we have hesitated to distribute them in case instructors want to use problems as assignments or test questions. These true–false quizzes provide students with immediate feedback as they read, without “giving away” answers to the multiple-choice questions and programming exercises.
- **DON’T DO IT:** This section at the end of each chapter summarizes common mistakes and pitfalls that plague new programmers while learning the current topic.
- **KEY TERMS:** Each chapter includes a list of newly introduced vocabulary, shown in the order of appearance in the text. The list of key terms provides a short review of the major concepts in the chapter.
- **SUMMARIES:** Following each chapter is a summary that recaps the programming concepts and techniques covered in the chapter. This feature provides a concise means for students to check their understanding of the main points in each chapter.
- **REVIEW QUESTIONS:** Each chapter includes 20 multiple-choice questions that serve as a review of chapter topics.
- **GAME ZONE:** Each chapter provides one or more exercises in which students can create interactive games using the programming techniques learned up to that point; 70 game programs are suggested in the book. The games are fun to create and play; writing them motivates students to master the necessary programming techniques. Students might exchange completed game programs with each other, suggesting improvements and discovering alternate ways to accomplish tasks.
- **CASES:** Each chapter contains two running case problems. These cases represent projects that continue to grow throughout a semester using concepts learned in each new chapter. Two cases allow instructors to assign different cases in alternate semesters or to divide students in a class into two case teams.

- **GLOSSARY:** This edition contains an alphabetized list of all key terms identified in the book, along with their definitions.
- **APPENDICES:** This edition includes useful appendices on working with the Java platform, data representation, formatting output, generating random numbers, and creating Javadoc comments.
- **QUALITY:** Every program example, exercise, and game solution was tested by the author and then tested again by a quality assurance team using Java Standard Edition (SE) 8, the most recent version available.

CourseMate

The more you study, the better the results. Make the most of your study time by accessing everything you need to succeed in one place. Read your textbook, take notes, review flashcards, watch videos, and take practice quizzes online. CourseMate goes beyond the book to deliver what you need! Learn more at www.cengage.com/coursemate.

The *Java Programming* CourseMate includes:

- **Debugging Exercises:** Four error-filled programs accompany each chapter. By debugging these programs, students can gain expertise in program logic in general and the Java programming language in particular.
- **Video Lessons:** Each chapter is accompanied by at least three video lessons that help to explain important chapter concepts. These videos were created and narrated by the author.
- **Interactive Study Aids:** An interactive eBook, quizzes, flashcards, and more!

Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly at www.CengageBrain.com.

Instructor Resources

The following teaching tools are available for download at our Instructor Companion Site. Simply search for this text at sso.cengage.com. An instructor login is required.

- **Electronic Instructor's Manual:** The Instructor's Manual that accompanies this textbook contains additional instructional material to assist in class preparation, including items such as Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A sample syllabus is also available. Additional exercises in the Instructor's Manual include:
 - **Tough Questions:** Two or more fairly difficult questions that an applicant might encounter in a technical job interview accompany each chapter. These questions are often open-ended; some involve coding and others might involve research.

- **Up for Discussion:** A few thought-provoking questions concerning programming in general or Java in particular supplement each chapter. The questions can be used to start classroom or online discussions, or to develop and encourage research, writing, and language skills.
- **Programming Exercises and Solutions:** Each chapter is accompanied by several programming exercises to supplement those offered in the text. Instructors can use these exercises as additional or alternate assignments, or as the basis for lectures.
- **Test Bank:** Cengage Learning Testing Powered by Cognero is a flexible, online system that allows you to:
 - Author, edit, and manage test bank content from multiple Cengage Learning solutions.
 - Create multiple test versions in an instant.
 - Deliver tests from your LMS, your classroom, or anywhere you want.
- **PowerPoint Presentations:** This text provides PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts. Files are provided for every figure in the text. Instructors may use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.
- **Solutions:** Solutions to “You Do It” exercises and all end-of-chapter exercises are available. Annotated solutions are provided for some of the multiple-choice Review Questions. For example, if students are likely to debate answer choices or not understand the choice deemed to be the correct one, a rationale is provided.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, including Dan Seiter, Development Editor; Alyssa Pratt, Senior Content Developer; Carmel Isaac, Content Project Manager; and Chris Scriver and Danielle Shaw, quality assurance testers. I am lucky to work with these professionals who are dedicated to producing high-quality instructional materials.

I am also grateful to the reviewers who provided comments and encouragement during this book’s development, including Bernice Cunningham, Wayne County Community College District; Bev Eckel, Iowa Western Community College; John Russo, Wentworth Institute of Technology; Leslie Spivey, Edison Community College; and Angeline Surber, Mesa Community College.

Thanks, too, to my husband, Geoff, for his constant support and encouragement. Finally, this book is dedicated to the newest Farrell, coming March 2015. As this book goes to production, I don’t know your name or even your gender, but I do know that I love you.

Joyce Farrell

Read This Before You Begin

xxvi

The following information will help you as you prepare to use this textbook.

To the User of the Data Files

To complete the steps and projects in this book, you need data files that have been created specifically for this book. Your instructor will provide the data files to you. You also can obtain the files electronically from *www.CengageBrain.com*. Find the ISBN of your title on the back cover of your book, then enter the ISBN in the search box at the top of the Cengage Brain home page. You can find the data files on the product page that opens. Note that you can use a computer in your school lab or your own computer to complete the exercises in this book.

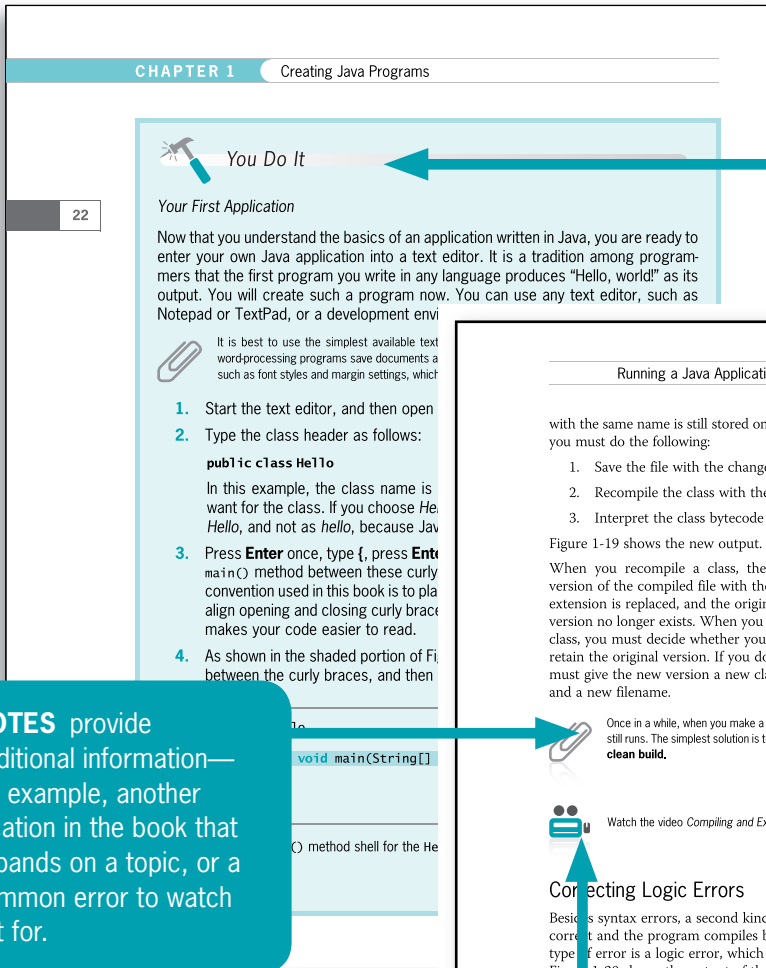
Using Your Own Computer

To use your own computer to complete the steps and exercises, you need the following:

- **Software:** Java SE 8, available from *www.oracle.com/technetwork/java/index.html*. Although almost all of the examples in this book will work with earlier versions of Java, this book was created using Java 8. The book clearly points out the few cases when an example is based on Java 7 and will not work with earlier versions of Java. You also need a text editor, such as Notepad. A few exercises ask you to use a browser for research.
- **Hardware:** If you are using Windows 8, the Java Web site suggests at least 128 MB of memory and at least 181 MB of disk space. For other operating system requirements, see *http://java.com/en/download/help*.

Features

This text focuses on helping students become better programmers and understand Java program development through a variety of key features. In addition to Chapter Objectives, Summaries, and Key Terms, these useful features will help students regardless of their learning styles.

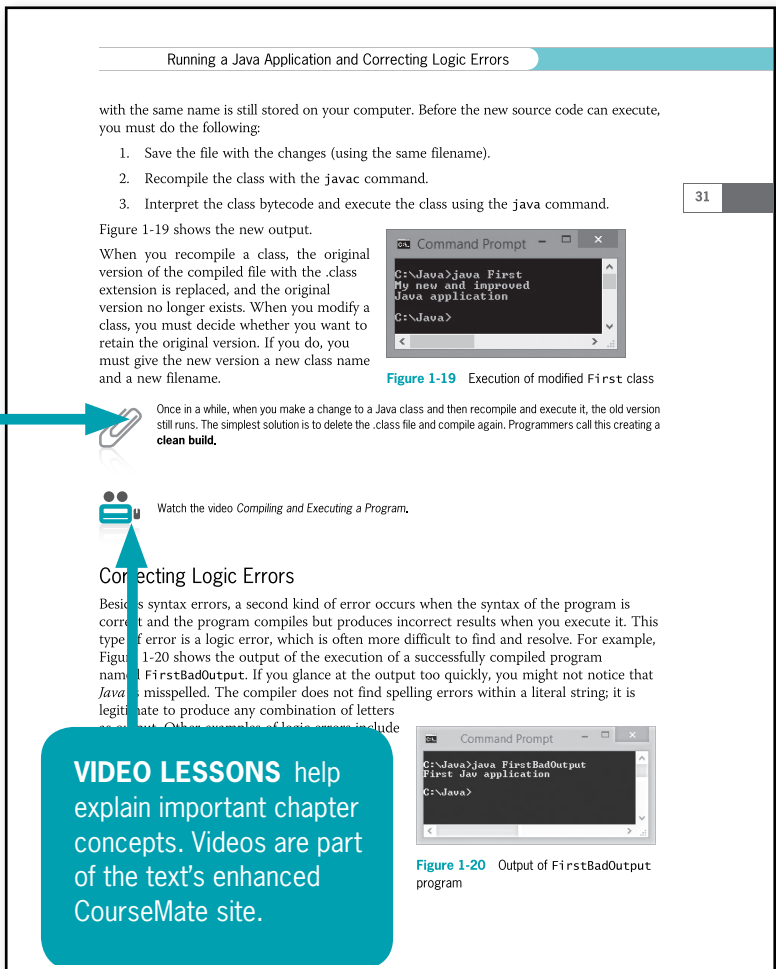


YOU DO IT sections walk students through program development step by step.

NOTES provide additional information—for example, another location in the book that expands on a topic, or a common error to watch out for.

The author does an awesome job: the examples, problems, and material are very easy to understand!

—Bernice Cunningham,
Wayne County Community
College District



VIDEO LESSONS help explain important chapter concepts. Videos are part of the text's enhanced CourseMate site.

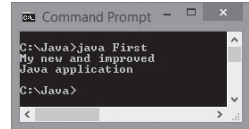


Figure 1-19 Execution of modified First class

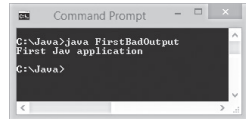


Figure 1-20 Output of FirstBadOutput program

```

Analyzing a Java Application that Produces Console Output

public class AnyClassName
{
    public static void main(String[] args)
    {
        /*****
    }
}
    
```

Figure 1-8 Shell code

Saving a Java Class

When you write a Java class, you must save it using a writable storage medium, such as a disk, DVD, or USB device. In Java, if a class is public (that is, if you use the `public` access specifier before the class name), you must save the class in a file with exactly the same name and a `.java` extension. For example, the `First` class must be stored in a file named `First.java`. The class name and filename must match exactly, including the use of uppercase and lowercase characters. If the extension is not `.java`, the Java compiler does not recognize the file as containing a Java class. Appendix A contains additional information on saving a Java application.

TWO TRUTHS & A LIE

Analyzing a Java Application that Produces Console Output

1. In the method header `public static void main(String[] args)`, the word `public` is an access specifier.
2. In the method header `public static void main(String[] args)`, the word `static` means that a method is accessible and usable, even though no objects of the class exist.
3. In the method header `public static void main(String[] args)`, the word `void` means that the `main()` method is an empty method.

The false statement is #3. In the method header `public static void main(String[] args)`, the word `void` means that the `main()` method does not return any value when it is called.

TWO TRUTHS & A LIE quizzes appear after each chapter section, with answers provided. The quiz contains three statements based on the preceding section of text—two statements are true and one is false. Answers give immediate feedback without “giving away” answers to the multiple-choice questions and programming problems later in the chapter. Students also have the option to take these quizzes electronically through the enhanced CourseMate site.

DON'T DO IT sections at the end of each chapter list advice for avoiding common programming errors.

Using the Scanner Class to Accept Keyboard Input

It is legal to write a single prompt that requests multiple input values—for example, “Please enter your age, area code, and zip code >>”. The user could then enter the three values separated with spaces, tabs, or Enter key presses. The values would be interpreted as separate tokens and could be retrieved with three separate `nextInt()` method calls. However, asking a user to enter multiple values often leads to mistakes. This book will follow the practice of using a separate prompt for each input value required.

Pitfall: Using `nextLine()` Following One of the Other Scanner Input Methods

You can encounter a problem when you use one of the numeric Scanner class retrieval methods or the `nextInt()` method before you use the `nextLine()` method. Consider the program in Figure 2-19. It is identical to the one in Figure 2-17, except that the user is asked for an age before being asked for a name. (See shading.) Figure 2-20 shows a typical execution.

```

import java.util.Scanner;
public class GetUserInfo2
{
    public static void main(String[] args)
    {
        String name;
        int age;
        Scanner inputDevice = new Scanner(System.in);
        System.out.print("Please enter your age >> ");
        age = inputDevice.nextInt();
        System.out.print("Please enter your name >> ");
        name = inputDevice.nextLine();
        System.out.println("Your name is " + name +
            " and you are " + age + " years old.");
    }
}
    
```

Figure 2-19 The `GetUserInfo2` class



Figure 2-20 Typical execution of the `GetUserInfo2` program

Save, compile, and execute the program. Now, the fractional portion of the result is omitted again. That's because the result of `sum / 2` is calculated first, and the result is an integer. Then, the whole-number result is cast to a double and assigned to a `double`—but the fractional part of the answer was already lost and casting is too late. Remove the newly added parentheses, save the program, compile it, and execute it again to confirm that the fractional part of the answer is reinstated.

As an alternative to the explicit cast in the division statement in the `ArithmeticDemo` program, you could write the average calculation as follows:

```

average = sum / 2.0;
    
```

In this calculation, when the integer `sum` is divided by the double constant `2.0`, the result is a double. The result then does not require any cast to be assigned to the double average without loss of data. Try this in your program.

Go to the Java Web site (www.oracle.com/technetwork/java/index.html), select **Java APIs**, and then select **Java SE 8**. Scroll through the list of **All Classes**, and select **PrintStream**, which is the data type for the `out` object used with the `println()` method. Scroll down to view the list of methods in the Method Summary. As you did in a previous exercise, notice the many versions of the `print()` and `println()` methods, including ones that accept a `String`, an `int`, and a `long`. Notice, however, that no versions accept a byte or a short. That's because when a byte or short is sent to the `print()` or `println()` method, it is automatically promoted to an `int`, so that version of the method is used.

Don't Do It

- Don't mispronounce “integer.” People who are unfamiliar with the term often say “integer,” inserting an extra *r*.
- Don't attempt to assign a literal constant floating-point number, such as `2.5`, to a `float` without following the constant with an uppercase or lowercase `F`. By default, constant floating-point values are doubles.
- Don't try to use a Java keyword as an identifier for a variable or constant. Table 1-1 in Chapter 1 contains a list of Java keywords.
- Don't attempt to assign a constant value under `-2,147,483,648` or over `+2,147,483,647` to a `long` variable without following the constant with an uppercase or lowercase `L`. By default, constant integers are `ints`, and a value under `-2,147,483,648` or over `+2,147,483,647` is too large to be an `int`.

THE DON'T DO IT ICON illustrates how NOT to do something—for example, having a dead code path in a program. This icon provides a visual jolt to the student, emphasizing that particular figures are NOT to be emulated and making students more careful to recognize problems in existing code.

Assessment

I found the author's explanation of difficult topics to be very clear and thorough.

—Leslie Spivey,
Edison Community College

PROGRAMMING EXERCISES provide opportunities to practice concepts. These exercises increase in difficulty and allow students to explore each major programming concept presented in the chapter. Additional programming exercises are available in the Instructor's Resource Kit.

xxix

Review Questions

A sequence of characters enclosed within double quotation marks is a _____.

- a. symbolic string
- b. literal string
- c. prompt
- d. command

To create a `String` object, you can use the keyword _____ before the constructor call, but you are not required to use this format.

- a. object
- b. create
- c. char
- d. new

A `String` variable name is a _____.

- a. reference
- b. value
- c. constant
- d. literal

The term that programmers use to describe objects that cannot be changed is _____.

- a. irrevocable
- b. nonvolatile
- c. immutable
- d. stable

Suppose that you declare two `String` objects as:

```
String word1 = new String("happy");  
String word2;
```

When you ask a user to enter a value for `word2`, if the user enters `word1 == word2` is _____.

- a. true
- b. false
- c. illegal
- d. unknown

If you declare two `String` objects as:

```
String word1 = new String("happy");  
String word2 = new String("happy");
```

the value of `word1.equals(word2)` is _____.

- a. true
- b. false
- c. illegal
- d. unknown

The method that determines whether two `String` objects are equal, regardless of case, is _____.

- a. `equalsIgnoreCase()`
- b. `toUpperCase()`
- c. `equalsIgn`
- d. `equals()`

383

CHAPTER 3 Using Methods, Classes, and Objects

20. If you use the automatically supplied default constructor when you create an object, _____.

- a. numeric fields are set to 0 (zero)
- b. character fields are set to blank
- c. Boolean fields are set to true
- d. All of these are true.

174

Exercises

Programming Exercises

- Suppose that you have created a program with only the following variables.

```
int a = 5;  
int b = 6;
```

Suppose that you also have a method with the following header:

```
public static void mathMethod(int a)
```

Which of the following method calls are legal?

- a. `mathMethod(a)`;
- b. `mathMethod(b)`;
- c. `mathMethod(a + b)`;
- d. `mathMethod(a, b)`;
- e. `mathMethod(2361)`;
- f. `mathMethod(12.78)`;
- g. `mathMethod(29987L)`;
- h. `mathMethod()`;
- i. `mathMethod(x)`;
- j. `mathMethod(a / b)`;

- Suppose that you have created a program with only the following variables.

```
int age = 34;  
int weight = 180;  
double height = 5.9;
```


Suppose that you also have a method with the following header:

```
public static void calculate(int age, double size)
```

Which of the following method calls are legal?

- a. `calculate(age, weight)`;
- b. `calculate(age, height)`;
- c. `calculate(weight, height)`;
- d. `calculate(height, age)`;
- e. `calculate(45.5, 120)`;
- f. `calculate(12, 120.2)`;
- g. `calculate(age, size)`;
- h. `calculate(2, 3)`;
- i. `calculate(age)`;
- j. `calculate(weight, weight)`;

REVIEW QUESTIONS test student comprehension of the major ideas and techniques presented. Twenty questions follow each chapter.

 Case Problems

1. Carly's Catering provides meals for parties and special events. Write a program that prompts the user for the number of guests attending an event and then computes the total price, which is \$35 per person. Display the company motto with the border that you created in the `CarlysMotto2` class in Chapter 1, and then display the number of guests, price per guest, and total price. Also display a message that indicates `true` or `false` depending on whether the job is classified as a large event—an event with 50 or more guests. Save the file as `CarlysEventPrice.java`.
2. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. Write a program that prompts the user for the number of minutes he rented a piece of sports equipment. Compute the rental cost as \$40 per hour plus \$1 per additional minute. (You might have surmised already that this rate has a logical flaw, but for now, calculate rates as described here. You can fix the problem after you read the chapter on decision making.) Display Sammy's motto with the border that you created in the `SammysMotto2` class in Chapter 1. Then display the hours, minutes, and total price. Save the file as `SammysRentalPrice.java`.

CASE PROBLEMS provide opportunities to build more detailed programs that continue to incorporate increasing functionality throughout the book.

DEBUGGING EXERCISES are included with each chapter because examining programs critically and closely is a crucial programming skill. Students can download these exercises at www.CengageBrain.com and through the CourseMate available for this text. These files are also available to instructors through sso.cengage.com.



Debugging Exercises

1. Each of the following files in the Chapter05 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with `Fix`. For example, save `DebugFive1.java` as `FixDebugFive1.java`.
 - a. `DebugFive1.java`
 - b. `DebugFive2.java`
 - c. `DebugFive3.java`
 - d. `DebugFive4.java`



Game Zone

1. In Chapter 1, you created a class called `RandomGuess`. In this game, players guess a number, the application generates a random number, and players determine whether they were correct. Now that you can make decisions, modify the application so it allows a player to enter a guess before the random number is displayed, and then displays a message indicating whether the player's guess was correct, too high, or too low. Save the file as `RandomGuess2.java`. (After you finish the next chapter, you will be able to modify the application so that the user can continue to guess until the correct answer is entered.)
2. Create a lottery game application. Generate three random numbers (see Appendix D for help in doing so), each between 0 and 9. Allow the user to guess three numbers. Compare each of the user's guesses to the three random numbers and display a message that includes the user's guess, the randomly determined three-digit number, and the amount of money the user has won as follows:

Matching Numbers	Award (\$)
Any one matching	10
Two matching	100
Three matching, not in order	1,000
Three matching in exact order	1,000,000
No matches	0

Make certain that your application accommodates repeating digits. For example, if a user guesses 1, 2, and 3, and the randomly generated digits are 1, 1, and 1, do not give the user credit for three correct guesses—just one. Save the file as `Lottery.java`.

GAME ZONE EXERCISES are included at the end of each chapter. Students can create games as an additional entertaining way to understand key programming concepts.

Creating Java Programs

In this chapter, you will:

- ⦿ Define basic programming terminology
- ⦿ Compare procedural and object-oriented programming
- ⦿ Describe the features of the Java programming language
- ⦿ Analyze a Java application that produces console output
- ⦿ Compile a Java class and correct syntax errors
- ⦿ Run a Java application and correct logic errors
- ⦿ Add comments to a Java class
- ⦿ Create a Java application that produces GUI output
- ⦿ Find help

Learning Programming Terminology

A **computer program** is a set of instructions that you write to tell a computer what to do. Computer equipment, such as a monitor or keyboard, is **hardware**, and programs are **software**. A program that performs a task for a user (such as calculating and producing paychecks, word processing, or playing a game) is **application software**; a program that manages the computer itself (such as Windows or Linux) is **system software**. The **logic** behind any computer program, whether it is an application or system program, determines the exact order of instructions needed to produce desired results. Much of this book describes how to develop the logic to create application software.

All computer programs ultimately are converted to machine language. **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute. Each type of processor (the internal hardware that handles computer instructions) has its own set of machine language instructions. Programmers often describe machine language using 1s and 0s to represent the on-and-off circuitry of computer systems.



The system that uses only 1s and 0s is the *binary numbering system*. Appendix B describes the binary system in detail. Later in this chapter, you will learn that *bytecode* is the name for the binary code created when Java programs are converted to machine language.

Machine language is a **low-level programming language**, or one that corresponds closely to a computer processor's circuitry. Low-level languages require you to use memory addresses for specific machines when you create commands. This means that low-level languages are difficult to use and must be customized for every type of machine on which a program runs.

Fortunately, programming has evolved into an easier task because of the development of high-level programming languages. A **high-level programming language** allows you to use a vocabulary of reasonable terms, such as *read*, *write*, or *add*, instead of the sequences of 1s and 0s that perform these tasks. High-level languages also allow you to assign single-word, intuitive names to areas of computer memory where you store data. This means you can use identifiers such as `hoursWorked` or `rateOfPay`, rather than having to remember their memory locations. Currently, over 2,000 high-level programming languages are available to developers; Java is one of them.

Each high-level language has its own **syntax**, or rules about how language elements are combined correctly to produce usable statements. For example, depending on the specific high-level language, you might use the verb *print* or *write* to produce output. All languages have a specific, limited vocabulary (the language's **keywords**) and a specific set of rules for using that vocabulary. When you are learning a computer programming language, such as Java, C++, or Visual Basic, you really are learning the vocabulary and syntax for that language.

Using a programming language, programmers write a series of **program statements**, similar to English sentences, to carry out the tasks they want the program to perform. Program statements are also known as **commands** because they are orders to the computer, such as “output this word” or “add these two numbers.”

After the program statements are written, high-level language programmers use a computer program called a **compiler** or **interpreter** to translate their language statements into machine language. A compiler translates an entire program before carrying out any statements, or **executing** them, whereas an interpreter translates one program statement at a time, executing a statement as soon as it is translated.



Whether you use a compiler or interpreter often depends on the programming language you use. For example, C++ is a compiled language, and Visual Basic is an interpreted language. Each type of translator has its supporters; programs written in compiled languages execute more quickly, whereas programs written in interpreted languages can be easier to develop and debug. Java uses the best of both technologies: a compiler to translate your programming statements and an interpreter to read the compiled code line by line when the program executes (also called **at run time**).

Compilers and interpreters issue one or more error messages each time they encounter an invalid program statement—that is, a statement containing a **syntax error**, or misuse of the language. Examples of syntax errors include misspelling a keyword or omitting a word that a statement requires. When a syntax error is detected, the programmer can correct the error and attempt another translation. Repairing all syntax errors is the first part of the process of **debugging** a program—freeing the program of all flaws or errors, also known as **bugs**. Figure 1-1 illustrates the steps a programmer takes while developing an executable program. You will learn more about debugging Java programs later in this chapter.

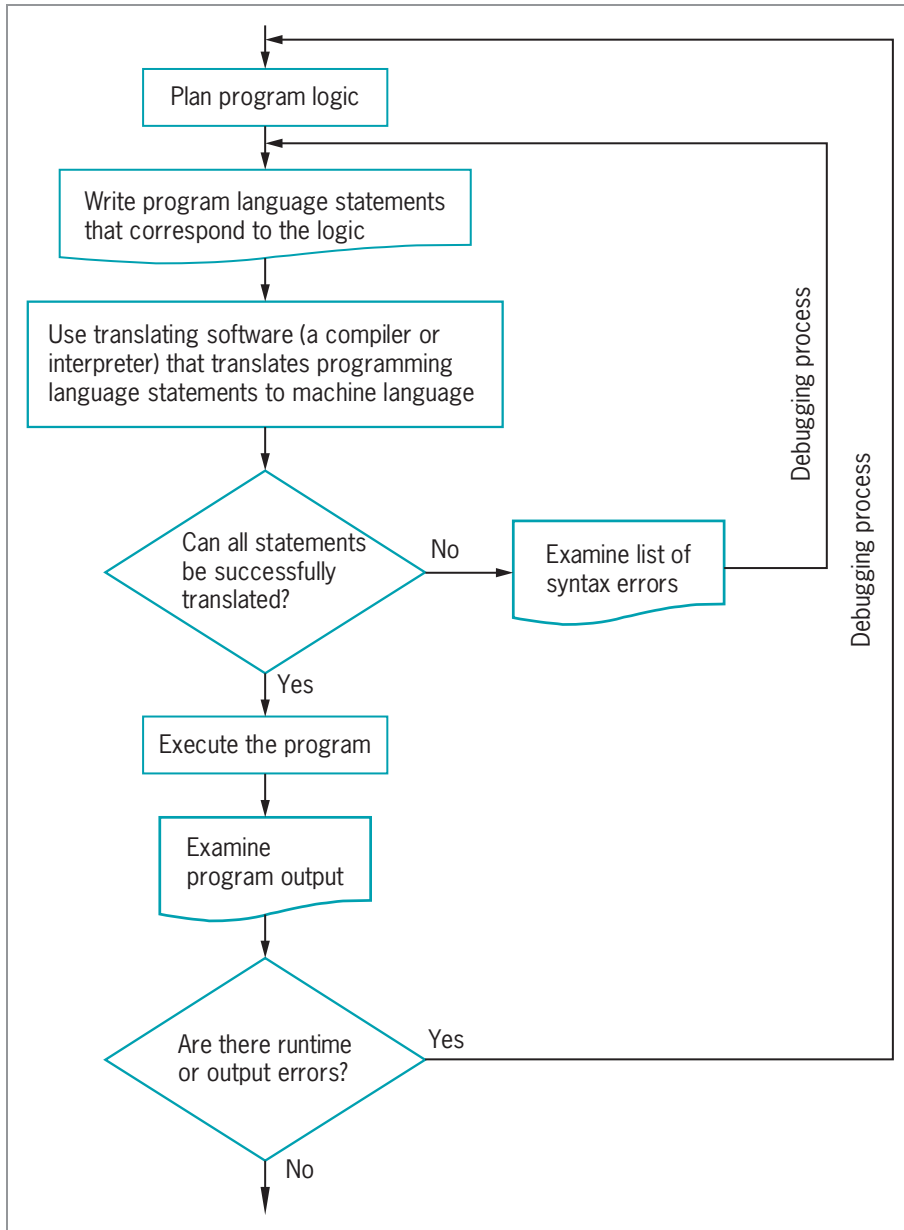


Figure 1-1 The program development process

As Figure 1-1 shows, you might write a program with correct syntax that still contains logic errors. A **logic error** is a bug that allows a program to run, but that causes it to operate incorrectly. Correct logic requires that all the right commands be issued in the appropriate order. Examples of logic errors include multiplying two values when you meant to divide

them or producing output prior to obtaining the appropriate input. When you develop a program of any significant size, you should plan its logic before you write any program statements.

Correcting logic errors is much more difficult than correcting syntax errors. Syntax errors are discovered by the language translator when you compile a program, but a program can be free of syntax errors and execute while still retaining logic errors. Often you can identify logic errors only when you examine a program's output. For example, if you know an employee's paycheck should contain the value \$4,000, but when you examine a payroll program's output you see that it holds \$40, then a logic error has occurred. Perhaps an incorrect calculation was performed, or maybe the hours worked value was output by mistake instead of the net pay value. When output is incorrect, the programmer must carefully examine all the statements within the program, revise or move the offending statements, and translate and test the program again.



Just because a program produces correct output does not mean it is free from logic errors. For example, suppose that a program should multiply two values entered by the user, that the user enters two 2s, and the output is 4. The program might actually be adding the values by mistake. The programmer would discover the logic error only by entering different values, such as 5 and 7, and examining the result.



Programmers call some logic errors **semantic errors**. For example, if you misspell a programming language word, you commit a syntax error, but if you use a correct word in the wrong context, you commit a semantic error.

TWO TRUTHS & A LIE

Learning Programming Terminology

In each "Two Truths & a Lie" section, two of the numbered statements are true, and one is false. Identify the false statement and explain why it is false.

1. Unlike a low-level programming language, a high-level programming language allows you to use a vocabulary of reasonable terms instead of the sequences of on-and-off switches that perform the corresponding tasks.
2. A syntax error occurs when you misuse a language; locating and repairing all syntax errors is part of the process of debugging a program.
3. Logic errors are fairly easy to find because the software that translates a program finds all the logic errors for you.

The false statement is #3. A language translator finds syntax errors, but logic errors can usually be discovered only by examining a program's output.

Comparing Procedural and Object-Oriented Programming Concepts

Two popular approaches to writing computer programs are procedural programming and object-oriented programming.

6

Procedural Programming

Procedural programming is a style of programming in which operations are executed one after another in sequence. In procedural applications, you create names for computer memory locations that can hold values—for example, numbers and text—in electronic form. The named computer memory locations are called **variables** because they hold values that might vary. For example, a payroll program might contain a variable named `rateOfPay`. The memory location referenced by the name `rateOfPay` might contain different values (a different value for every employee of the company) at different times. During the execution of the payroll program, each value stored under the name `rateOfPay` might have many operations performed on it—for example, the value might be read from an input device, be multiplied by another variable representing hours worked, and be printed on paper.

For convenience, the individual operations used in a computer program are often grouped into logical units called **procedures**. For example, a series of four or five comparisons and calculations that together determine a person's federal withholding tax value might be grouped as a procedure named `calculateFederalWithholding`. A procedural program defines the variable memory locations and then calls a series of procedures to input, manipulate, and output the values stored in those locations. When a program **calls a procedure**, the current logic is temporarily abandoned so that the procedure's commands can execute. A single procedural program often contains hundreds of variables and procedure calls. Procedures are also called *modules*, *methods*, *functions*, and *subroutines*. Users of different programming languages tend to use different terms. As you will learn later in this chapter, Java programmers most frequently use the term *method*.

Object-Oriented Programming

Object-oriented programming is an extension of procedural programming in which you take a slightly different approach to writing computer programs. Writing **object-oriented programs** involves:

- Creating classes, which are blueprints for objects
- Creating objects, which are specific instances of those classes
- Creating applications that manipulate or use those objects



Programmers use *OO* as an abbreviation for *object-oriented*; it is pronounced “oh oh.” Object-oriented programming is abbreviated *OOP*, and pronounced to rhyme with *soup*.

Originally, object-oriented programming was used most frequently for two major types of applications:

- **Computer simulations**, which attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate
- **Graphical user interfaces**, or **GUIs** (pronounced “gooeys”), which allow users to interact with a program in a graphical environment

Thinking about objects in these two types of applications makes sense. For example, a city might want to develop a program that simulates traffic patterns to help prevent traffic tie-ups. Programmers would create classes for objects such as cars and pedestrians that contain their own data and rules for behavior. For example, each car has a speed and a method for changing that speed. The specific instances of cars could be set in motion to create a simulation of a real city at rush hour.

Creating a GUI environment for users is also a natural use for object orientation. It is easy to think of the components a user manipulates on a computer screen, such as buttons and scroll bars, as similar to real-world objects. Each GUI object contains data—for example, a button on a screen has a specific size and color. Each object also contains behaviors—for example, each button can be clicked and reacts in a specific way when clicked. Some people consider the term *object-oriented programming* to be synonymous with GUI programming, but object-oriented programming means more. Although many GUI programs are object oriented, not all object-oriented programs use GUI objects. Modern businesses use object-oriented design techniques when developing all sorts of business applications, whether they are GUI applications or not. In the first 13 chapters of this book, you will learn object-oriented techniques that are appropriate for any program type; in the last chapters, you will apply what you have learned about those techniques specifically to GUI applications.

Understanding object-oriented programming requires grasping three basic concepts:

- Encapsulation as it applies to classes as objects
- Inheritance
- Polymorphism

Understanding Classes, Objects, and Encapsulation

In object-oriented terminology, a **class** is a term that describes a group or collection of objects with common properties. In the same way that a blueprint exists before any houses are built from it, and a recipe exists before any cookies are baked from it, a class definition exists before any objects are created from it. A **class definition** describes what attributes its objects will have and what those objects will be able to do. **Attributes** are the characteristics that define an object; they are **properties** of the object. When you learn a programming language such as Java, you learn to work with two types of classes: those that have already been developed by the language’s creators and your own new, customized classes.

An **object** is a specific, concrete **instance** of a class. Creating an instance is called **instantiation**. You can create objects from classes that you write and from classes written by other programmers, including Java's creators. The values contained in an object's properties often differentiate instances of the same class from one another. For example, the class `Automobile` describes what `Automobile` objects are like. Some properties of the `Automobile` class are `make`, `model`, `year`, and `color`. Each `Automobile` object possesses the same attributes, but not necessarily the same values for those attributes. One `Automobile` might be a 2010 white Ford Taurus and another might be a 2015 red Chevrolet Camaro. Similarly, your dog has the properties of all `Dogs`, including a breed, name, age, and whether its shots are current. The values of the properties of an object are referred to as the object's **state**. In other words, you can think of objects as roughly equivalent to nouns, and of their attributes as similar to adjectives that describe the nouns.

When you understand an object's class, you understand the characteristics of the object. If your friend purchases an `Automobile`, you know it has a model name, and if your friend gets a `Dog`, you know the dog has a breed. Knowing what attributes exist for classes allows you to ask appropriate questions about the states or values of those attributes. For example, you might ask how many miles the car gets per gallon, but you would not ask whether the car has had shots. Similarly, in a GUI operating environment, you expect each component to have specific, consistent attributes and methods, such as a window having a title bar and a close button, because each component gains these properties as a member of the general class of GUI components. Figure 1-2 shows the relationship of some `Dog` objects to the `Dog` class.



By convention, programmers using Java begin their class names with an uppercase letter. Thus, the class that defines the attributes and methods of an automobile would probably be named `Automobile`, and the class for dogs would probably be named `Dog`. However, following this convention is not required to produce a workable program.

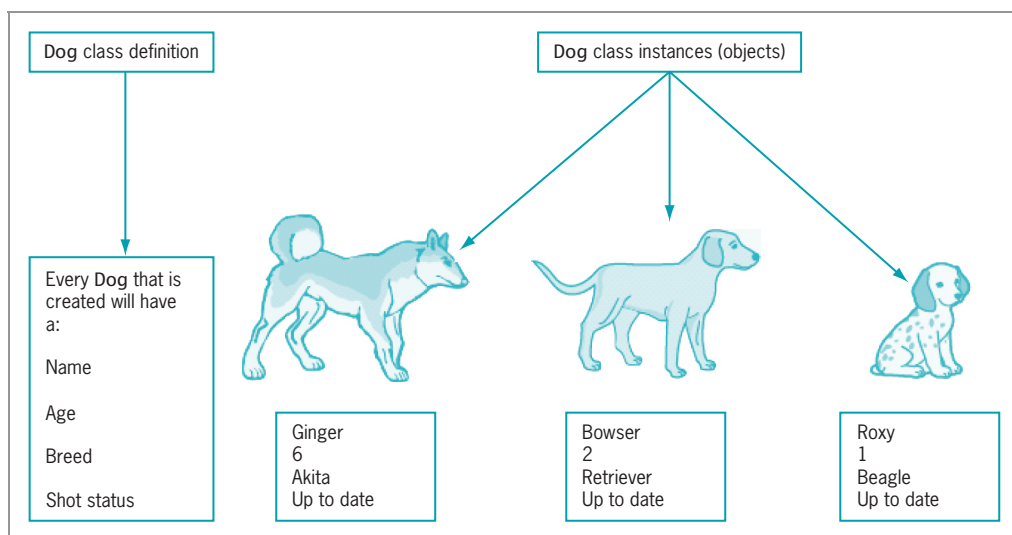


Figure 1-2 Dog class definition and some objects created from it

Besides defining properties, classes define methods their objects can use. A **method** is a self-contained block of program code that carries out some action, similar to a procedure in a procedural program. An `Automobile`, for example, might have methods for moving forward, moving backward, and determining the status of its gas tank. Similarly, a `Dog` might have methods for walking, eating, and determining its name, and a program's GUI components might have methods for maximizing and minimizing them as well as determining their size. In other words, if objects are similar to nouns, then methods are similar to verbs.

In object-oriented classes, attributes and methods are encapsulated into objects.

Encapsulation refers to two closely related object-oriented notions:

- Encapsulation is the enclosure of data and methods within an object. Encapsulation allows you to treat all of an object's methods and data as a single entity. Just as an actual dog contains all of its attributes and abilities, so would a program's `Dog` object.
- Encapsulation also refers to the concealment of an object's data and methods from outside sources. Concealing data is sometimes called *information hiding*, and concealing how methods work is *implementation hiding*; you will learn more about both terms in the chapter "Using Methods, Classes, and Objects." Encapsulation lets you hide specific object attributes and methods from outside sources and provides the security that keeps data and methods safe from inadvertent changes.

If an object's methods are well written, the user can be unaware of the low-level details of how the methods are executed, and the user must simply understand the interface or interaction between the method and the object. For example, if you can fill your `Automobile` with gasoline, it is because you understand the interface between the gas pump nozzle and the vehicle's gas tank opening. You don't need to understand how the pump works mechanically or where the gas tank is located inside your vehicle. If you can read your speedometer, it does not matter how the displayed figure is calculated. As a matter of fact, if someone produces a superior, more accurate speed-determining device and inserts it in your `Automobile`, you don't have to know or care how it operates, as long as your interface remains the same. The same principles apply to well-constructed classes used in object-oriented programs—programs that use classes only need to work with interfaces.

Understanding Inheritance and Polymorphism

An important feature of object-oriented program design is **inheritance**—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. For example, `Automobile` is a class, and all `Automobile` objects share many traits and abilities. `Convertible` is a class that inherits from the `Automobile` class; a `Convertible` is a type of `Automobile` that has and can do everything a "plain" `Automobile` does—but with an added ability to lower its top. (In turn, `Automobile` inherits from the `Vehicle` class.)

`Convertible` is not an object—it is a class. A specific `Convertible` is an object—for example, `my1967BlueMustangConvertible`.

Inheritance helps you understand real-world objects. For example, the first time you encounter a convertible, you already understand how the ignition, brakes, door locks, and