Introduction to JAVA™

PROGRAMMING AND DATA STRUCTURES

COMPREHENSIVE VERSION

Y. DANIEL LIANG

Pearson

12th Edition

INTRODUCTION TO JAVA™ PROGRAMMING AND DATA STRUCTURES

COMPREHENSIVE VERSION

Twelfth Edition

Y. Daniel Liang

Georgia Southern University

Pearson

To Samantha, Michael, and Michelle

Library of Congress Cataloging-in-Publication Data

Names: Liang, Y. Daniel, author.
Title: Java programming and data structures / Y. Daniel Liang, Georgia Southern University.
Other titles: Introduction to Java programming and data structures
Description: Twelfth edition. Comprehensive version | Hoboken, NJ : Pearson, 2019. | Revised edition of: Introduction to Java programming and data structures / Y. Daniel Liang, Georgia Southern University. Eleventh edition. Comprehensive version. 2018. | Includes bibliographical references and index.
Identifiers: LCCN 2019038073 | ISBN 9780136520238 (paperback)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 L52 2019 | DDC 005.13/3–dc23
LC record available at https://lccn.loc.gov/2019038073

ScoutAutomatedPrintCode

LLE ISBN
ISBN-10: 0-13-651996-2
ISBN-13: 978-0-13-651996-6

SE
ISBN-10: 0-13-652023-5
ISBN-13: 978-0-13-652023-8

PREFACE

Dear Reader,

Many of you have provided feedback on earlier editions of this book, and your comments and suggestions have greatly improved the book. This edition has been substantially enhanced in presentation, organization, examples, exercises, and supplements.

fundamentals-first

The book is fundamentals first by introducing basic programming concepts and techniques before designing custom classes. The fundamental concepts and techniques of selection statements, loops, methods, and arrays are the foundation for programming. Building this strong foundation prepares students to learn object-oriented programming and advanced Java programming.

problem-driven

This book teaches programming in a problem-driven way that focuses on problem solving rather than syntax. We make introductory programming interesting by using thought-provoking problems in a broad context. The central thread of early chapters is on problem solving. Appropriate syntax and library are introduced to enable readers to write programs for solving the problems. To support the teaching of programming in a problem-driven way, the book provides a wide variety of problems at various levels of difficulty to motivate students. To appeal to students in all majors, the problems cover many application areas, including math, science, business, financial, gaming, animation, and multimedia.

data structures

The book seamlessly integrates programming, data structures, and algorithms into one text. It employs a practical approach to teach data structures. We first introduce how to use various data structures to develop efficient algorithms, and then show how to implement these data structures. Through implementation, students gain a deep understanding on the efficiency of data structures and on how and when to use certain data structures. Finally, we design and implement custom data structures for trees and graphs.

comprehensive version

brief version

AP Computer Science

The book is widely used in the introductory programming, data structures, and algorithms courses in the universities around the world. This *comprehensive version* covers fundamentals of programming, object-oriented programming, GUI programming, data structures, algorithms, concurrency, networking, database, and Web programming. It is designed to prepare students to become proficient Java programmers. A *brief version* (*Introduction to Java Programming*, Brief Version, Twelfth Edition) is available for a first course on programming, commonly known as CS1. The brief version contains the first 18 chapters of the comprehensive version. An AP version of the book is also available for high school students taking an AP Computer Science course.

examples and exercises

The best way to teach programming is *by example*, and the only way to learn programming is *by doing*. Basic concepts are explained by example and a large number of exercises with various levels of difficulty are provided for students to practice. For our programming courses, we assign programming exercises after each lecture.

Our goal is to produce a text that teaches problem solving and programming in a broad context using a wide variety of interesting examples. If you have any comments on and suggestions for improving the book, please email me.

Sincerely,

Y. Daniel Liang
y.daniel.liang@gmail.com
www.pearsonhighered.com/liang

ACM/IEEE Curricular 2013 and ABET Course Assessment

The new ACM/IEEE Computer Science Curricular 2013 defines the Body of Knowledge organized into 18 Knowledge Areas. To help instructors design the courses based on this book, we provide sample syllabi to identify the Knowledge Areas and Knowledge Units. The sample syllabi are for a three semester course sequence and serve as an example for institutional customization. The sample syllabi are accessible from the Instructor Resource Website.

Many of our users are from the ABET-accredited programs. A key component of the ABET accreditation is to identify the weakness through continuous course assessment against the course outcomes. We provide sample course outcomes for the courses and sample exams for measuring course outcomes on the Instructor Resource Website.

What's New in This Edition?

This edition is completely revised in every detail to enhance clarity, presentation, content, examples, and exercises. The major improvements are as follows:

- Updated to Java 9, 10, and 11. Examples are improved and simplified by using the new features in Java 9, 10, 11.
- The GUI chapters are updated to JavaFX 11. The examples are revised. The user interfaces in the examples and exercises are now resizable and displayed in the center of the window.
- More examples and exercises in the data structures chapters use Lambda expressions to simplify coding.
- Both `Comparable` and `Comparator` are used to compare elements in `Heap`, `PriorityQueue`, `BST`, and `AVLTree`. This is consistent with the Java API and is more useful and flexible.
- String matching algorithms are introduced in Chapter 22.
- VideoNotes are updated.
- Provided additional exercises not printed in the book. These exercises are available for instructors only.

Please visit www.pearsonhighered.com/liang for a complete list of new features as well as correlations to the previous edition.

Pedagogical Features

The book uses the following elements to help students get the most from the material:

- The **Objectives** at the beginning of each chapter list what students should learn from the chapter. This will help them determine whether they have met the objectives after completing the chapter.
- The **Introduction** opens the discussion with a thought-provoking question to motivate the reader to delve into the chapter.
- **Key Points** highlight the important concepts covered in each section.
- **Check Points** provide review questions to help students track their progress as they read through the chapter and evaluate their learning.

- **Problems and Case Studies**, carefully chosen and presented in an easy-to-follow style, teach problem solving and programming concepts. The book uses many small, simple, and stimulating examples to demonstrate important ideas.
- The **Chapter Summary** reviews the important subjects that students should understand and remember. It helps them reinforce the key concepts they have learned in the chapter.
- **Quizzes** are accessible online, grouped by sections, for students to do self-test on programming concepts and techniques.
- **Programming Exercises** are grouped by sections to provide students with opportunities to apply the new skills they have learned on their own. The level of difficulty is rated as easy (no asterisk), moderate (*), hard (**), or challenging (***). The trick of learning programming is practice, practice, and practice. To that end, the book provides a great many exercises. Additionally, more than 200 programming exercises with solutions are provided to the instructors on the Instructor Resource Website. These exercises are not printed in the text.
- **Notes**, **Tips**, **Cautions**, and **Design Guides** are inserted throughout the text to offer valuable advice and insight on important aspects of program development.

Note
Provides additional information on the subject and reinforces important concepts.

Tip
Teaches good programming style and practice.

Caution
Helps students steer away from the pitfalls of programming errors.

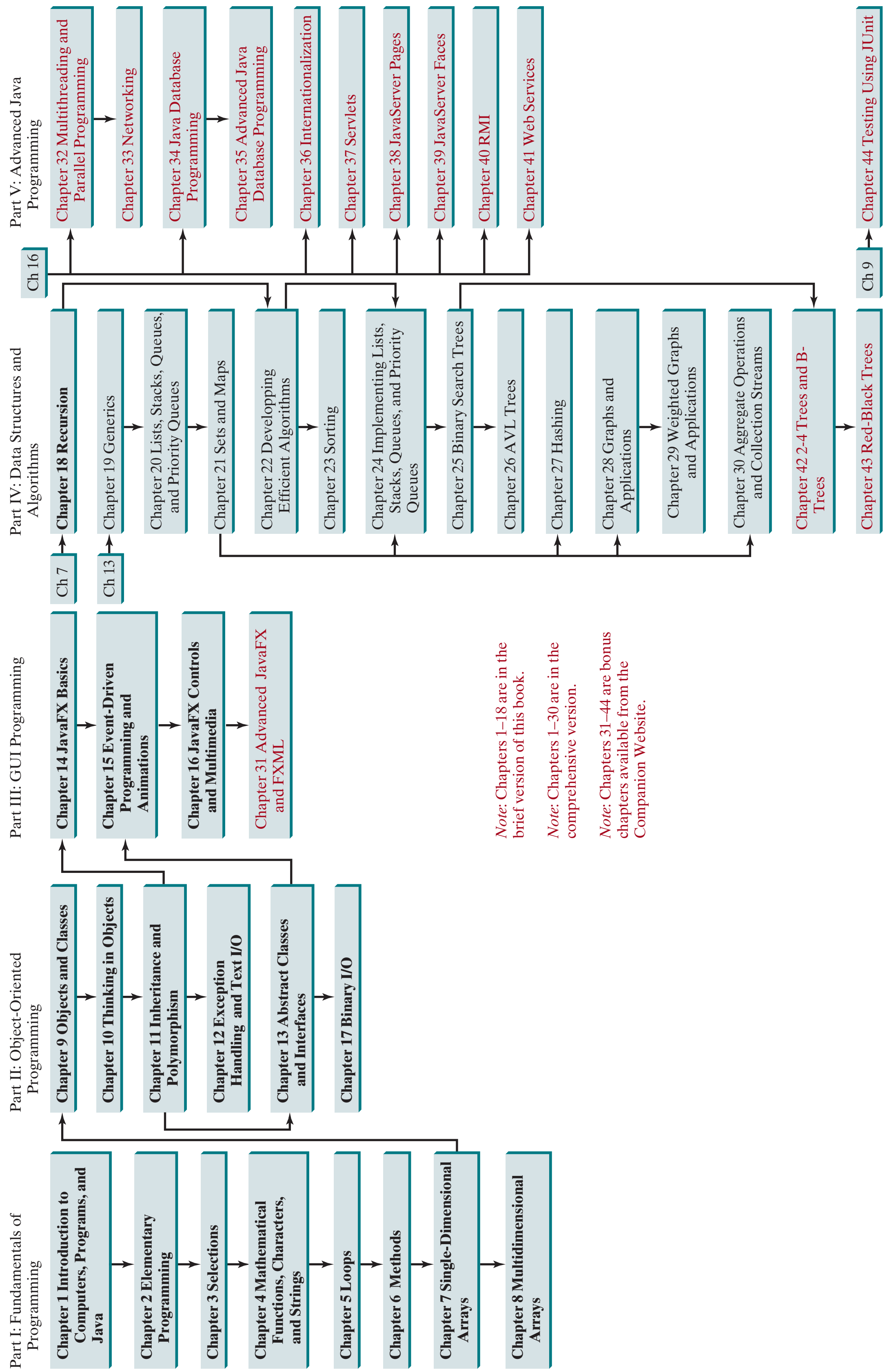
Design Guide
Provides guidelines for designing programs.

Flexible Chapter Orderings

The book is designed to provide flexible chapter orderings to enable GUI, exception handling, recursion, generics, and the Java Collections Framework to be covered earlier or later. The diagram on the next page shows the chapter dependencies.

Organization of the Book

The chapters can be grouped into five parts that, taken together, form a comprehensive introduction to Java programming, data structures and algorithms, and database and Web programming. Because knowledge is cumulative, the early chapters provide the conceptual basis for understanding programming and guide students through simple examples and exercises; subsequent chapters progressively present Java programming in detail, culminating with the development of comprehensive Java applications. The appendixes contain a mixed bag of topics, including an introduction to number systems, bitwise operations, regular expressions, and enumerated types.


Part I: Fundamentals of Programming
Chapter 1 Introduction to Computers, Programs, and Java
Chapter 2 Elementary Programming
Chapter 3 Selections
Chapter 4 Mathematical Functions, Characters, and Strings
Chapter 5 Loops
Chapter 6 Methods
Chapter 7 Single-Dimensional Arrays
Chapter 8 Multidimensional Arrays
Part II: Object-Oriented Programming
Chapter 9 Objects and Classes
Chapter 10 Thinking in Objects
Chapter 11 Inheritance and Polymorphism
Chapter 12 Exception Handling and Text I/O
Chapter 13 Abstract Classes and Interfaces
Chapter 17 Binary I/O
Part III: GUI Programming
Chapter 14 JavaFX Basics
Chapter 15 Event-Driven Programming and Animations
Chapter 16 JavaFX Controls and Multimedia
Chapter 31 Advanced JavaFX and FXML
Note: Chapters 1–18 are in the brief version of this book.
Note: Chapters 1–30 are in the comprehensive version.
Note: Chapters 31–44 are bonus chapters available from the Companion Website.
Ch 7
Ch 13
Part IV: Data Structures and Algorithms
Chapter 18 Recursion
Chapter 19 Generics
Chapter 20 Lists, Stacks, Queues, and Priority Queues
Chapter 21 Sets and Maps
Chapter 22 Developping Efficient Algorithms
Chapter 23 Sorting
Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues
Chapter 25 Binary Search Trees
Chapter 26 AVL Trees
Chapter 27 Hashing
Chapter 28 Graphs and Applications
Chapter 29 Weighted Graphs and Applications
Chapter 30 Aggregate Operations and Collection Streams
Chapter 42 2-4 Trees and B-Trees
Chapter 43 Red-Black Trees
Ch 16
Ch 9
Part V: Advanced Java Programming
Chapter 32 Multithreading and Parallel Programming
Chapter 33 Networking
Chapter 34 Java Database Programming
Chapter 35 Advanced Java Database Programming
Chapter 36 Internationalization
Chapter 37 Servlets
Chapter 38 JavaServer Pages
Chapter 39 JavaServer Faces
Chapter 40 RMI
Chapter 41 Web Services
Chapter 44 Testing Using JUnit

Part I: Fundamentals of Programming (Chapters 1–8)

The first part of the book is a stepping stone, preparing you to embark on the journey of learning Java. You will begin to learn about Java (Chapter 1) and fundamental programming techniques with primitive data types, variables, constants, assignments, expressions, and operators (Chapter 2), selection statements (Chapter 3), mathematical functions, characters, and strings (Chapter 4), loops (Chapter 5), methods (Chapter 6), and arrays (Chapters 7–8). After Chapter 7, you can jump to Chapter 18 to learn how to write recursive methods for solving inherently recursive problems.

Part II: Object-Oriented Programming (Chapters 9–13, and 17)

This part introduces object-oriented programming. Java is an object-oriented programming language that uses abstraction, encapsulation, inheritance, and polymorphism to provide great flexibility, modularity, and reusability in developing software. You will learn programming with objects and classes (Chapters 9–10), class inheritance (Chapter 11), polymorphism (Chapter 11), exception handling (Chapter 12), abstract classes (Chapter 13), and interfaces (Chapter 13). Text I/O is introduced in Chapter 12 and binary I/O is discussed in Chapter 17.

Part III: GUI Programming (Chapters 14–16 and Bonus Chapter 31)

JavaFX is a new framework for developing Java GUI programs. It is not only useful for developing GUI programs, but also an excellent pedagogical tool for learning object-oriented programming. This part introduces Java GUI programming using JavaFX in Chapters 14–16. Major topics include GUI basics (Chapter 14), container panes (Chapter 14), drawing shapes (Chapter 14), event-driven programming (Chapter 15), animations (Chapter 15), and GUI controls (Chapter 16), and playing audio and video (Chapter 16). You will learn the architecture of JavaFX GUI programming and use the controls, shapes, panes, image, and video to develop useful applications. Chapter 31 covers advanced features in JavaFX.

Part IV: Data Structures and Algorithms (Chapters 18–30 and Bonus Chapters 42–43)

This part covers the main subjects in a typical data structures and algorithms course. Chapter 18 introduces recursion to write methods for solving inherently recursive problems. Chapter 19 presents how generics can improve software reliability. Chapters 20 and 21 introduce the Java Collection Framework, which defines a set of useful API for data structures. Chapter 22 discusses measuring algorithm efficiency in order to choose an appropriate algorithm for applications. Chapter 23 describes classic sorting algorithms. You will learn how to implement several classic data structures lists, queues, and priority queues in Chapter 24. Chapters 25 and 26 introduce binary search trees and AVL trees. Chapter 27 presents hashing and implementing maps and sets using hashing. Chapters 28 and 29 introduce graph applications. Chapter 30 introduces aggregate operations for collection streams. The 2-4 trees, B-trees, and red-black trees are covered in Bonus Chapters 42–43.

Part V: Advanced Java Programming (Chapters 32-41, 44)

This part of the book is devoted to advanced Java programming. Chapter 32 treats the use of multithreading to make programs more responsive and interactive and introduces parallel programming. Chapter 33 discusses how to write programs that talk with each other from different hosts over the Internet. Chapter 34 introduces the use of Java to develop database projects. Chapter 35 delves into advanced Java database programming. Chapter 36 covers the use of internationalization support to develop projects for international audiences. Chapters 37 and 38 introduce how to use Java servlets and JavaServer Pages to generate dynamic content from Web servers. Chapter 39 introduces modern Web application development using JavaServer Faces. Chapter 40 introduces remote method invocation and Chapter 41 discusses Web services. Chapter 44 introduces testing Java programs using JUnit.

Appendixes

This part of the book covers a mixed bag of topics. Appendix A lists Java keywords. Appendix B gives tables of ASCII characters and their associated codes in decimal and in hex. Appendix C shows the operator precedence. Appendix D summarizes Java modifiers and their usage. Appendix E discusses special floating-point values. Appendix F introduces number systems and conversions among binary, decimal, and hex numbers. Finally, Appendix G introduces bitwise operations. Appendix H introduces regular expressions. Appendix I covers enumerated types.

Java Development Tools

You can use a text editor, such as the Windows Notepad or WordPad, to create Java programs and to compile and run the programs from the command window. You can also use a Java development tool, such as NetBeans or Eclipse. These tools support an integrated development environment (IDE) for developing Java programs quickly. Editing, compiling, building, executing, and debugging programs are integrated in one graphical user interface. Using these tools effectively can greatly increase your programming productivity. NetBeans and Eclipse are easy to use if you follow the tutorials. Tutorials on NetBeans and Eclipse can be found in the supplements on the Companion Website www.pearsonhighered.com/liang.

IDE tutorials

Student Resource Website

The Student Resource Website (www.pearsonhighered.com/liang) contains the following resources:

- Answers to CheckPoint questions
- Solutions to majority of even-numbered programming exercises
- Source code for the examples in the book
- Interactive quiz (organized by sections for each chapter)
- Supplements
- Debugging tips
- Video notes
- Algorithm animations
- Errata

Supplements

The text covers the essential subjects. The supplements extend the text to introduce additional topics that might be of interest to readers. The supplements are available from the Companion Website.

Instructor Resource Website

The Instructor Resource Website, accessible from www.pearsonhighered.com/liang, contains the following resources:

- Microsoft PowerPoint slides with interactive buttons to view full-color, syntax-highlighted source code and to run programs without leaving the slides.
- Solutions to majority of odd-numbered programming exercises.

- More than 200 additional programming exercises and 300 quizzes organized by chapters. These exercises and quizzes are available only to the instructors. Solutions to these exercises and quizzes are provided.
- Web-based quiz generator. (Instructors can choose chapters to generate quizzes from a large database of more than two thousand questions.)
- Sample exams. Most exams have four parts:
  - Multiple-choice questions or short-answer questions
  - Correct programming errors
  - Trace programs
  - Write programs
- Sample exams with ABET course assessment.
- Projects. In general, each project gives a description and asks students to analyze, design, and implement the project.

Some readers have requested the materials from the Instructor Resource Website. Please understand that these are for instructors only. Such requests will not be answered.

Online Practice and Assessment with MyProgrammingLab

MyProgrammingLab™

MyProgrammingLab helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students who often struggle with the basic concepts and paradigms of popular high-level programming languages.

A self-study and homework tool, a MyProgrammingLab course consists of hundreds of small practice problems organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of their code submissions and offers targeted hints that enable students to figure out what went wrong—and why. For instructors, a comprehensive gradebook tracks correct and incorrect answers and stores the code inputted by students for review.

MyProgrammingLab is offered to users of this book in partnership with Turing's Craft, the makers of the CodeLab interactive programming exercise system. For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

Video Notes

VideoNote

We are excited about the new Video Notes feature that is found in this new edition. These videos provide additional help by presenting examples of key topics and showing how to solve problems completely from design through coding. Video Notes are available from www.pearsonhighered.com/liang.

Algorithm Animations

Animation

We have provided numerous animations for algorithms. These are valuable pedagogical tools to demonstrate how algorithms work. Algorithm animations can be accessed from the Companion Website.

Acknowledgments

I would like to thank Georgia Southern University for enabling me to teach what I write and for supporting me in writing what I teach. Teaching is the source of inspiration for continuing to improve the book. I am grateful to the instructors and students who have offered comments, suggestions, corrections, and praise. My special thanks go to Stefan Andrei of Lamar University and William Bahn of University of Colorado Colorado Springs for their help to improve the data structures part of this book.

This book has been greatly enhanced thanks to outstanding reviews for this and previous editions. The reviewers are: Elizabeth Adams (James Madison University), Syed Ahmed (North Georgia College and State University), Omar Aldawud (Illinois Institute of Technology), Stefan Andrei (Lamar University), Yang Ang (University of Wollongong, Australia), Kevin Bierre (Rochester Institute of Technology), Aaron Braskin (Mira Costa High School), David Champion (DeVry Institute), James Chegwidden (Tarrant County College), Anup Dargar (University of North Dakota), Daryl Detrick (Warren Hills Regional High School), Charles Dierbach (Towson University), Frank Ducrest (University of Louisiana at Lafayette), Erica Eddy (University of Wisconsin at Parkside), Summer Ehresman (Center Grove High School), Deena Engel (New York University), Henry A. Etlinger (Rochester Institute of Technology), James Ten Eyck (Marist College), Myers Foreman (Lamar University), Olac Fuentes (University of Texas at El Paso), Edward F. Gehringer (North Carolina State University), Harold Grossman (Clemson University), Barbara Guillot (Louisiana State University), Stuart Hansen (University of Wisconsin, Parkside), Dan Harvey (Southern Oregon University), Ron Hofman (Red River College, Canada), Stephen Hughes (Roanoke College), Vladan Jovanovic (Georgia Southern University), Deborah Kabura Kariuki (Stony Point High School), Edwin Kay (Lehigh University), Larry King (University of Texas at Dallas), Nana Kofi (Langara College, Canada), George Koutsogiannakis (Illinois Institute of Technology), Roger Kraft (Purdue University at Calumet), Norman Krumpe (Miami University), Hong Lin (DeVry Institute), Dan Lipsa (Armstrong State University), James Madison (Rensselaer Polytechnic Institute), Frank Malinowski (Darton College), Tim Margush (University of Akron), Debbie Masada (Sun Microsystems), Blayne Mayfield (Oklahoma State University), John McGrath (J.P. McGrath Consulting), Hugh McGuire (Grand Valley State), Shyamal Mitra (University of Texas at Austin), Michel Mitri (James Madison University), Kenrick Mock (University of Alaska Anchorage), Frank Murgolo (California State University, Long Beach), Jun Ni (University of Iowa), Benjamin Nystuen (University of Colorado at Colorado Springs), Maureen Opkins (CA State University, Long Beach), Gavin Osborne (University of Saskatchewan), Kevin Parker (Idaho State University), Dale Parson (Kutztown University), Mark Pendergast (Florida Gulf Coast University), Richard Povinelli (Marquette University), Roger Priebe (University of Texas at Austin), Mary Ann Pumphrey (De Anza Junior College), Pat Roth (Southern Polytechnic State University), Amr Sabry (Indiana University), Ben Setzer (Kennesaw State University), Carolyn Schauble (Colorado State University), David Scuse (University of Manitoba), Ashraf Shirani (San Jose State University), Daniel Spiegel (Kutztown University), Joslyn A. Smith (Florida Atlantic University), Lixin Tao (Pace University), Ronald F. Taylor (Wright State University), Russ Tront (Simon Fraser University), Deborah Trytten (University of Oklahoma), Michael Verdicchio (Citadel), Kent Vidrine (George Washington University), and Bahram Zartoshty (California State University at Northridge).

It is a great pleasure, honor, and privilege to work with Pearson. I would like to thank Tracy Johnson and her colleagues Marcia Horton, Demetrius Hall, Yvonne Vannatta, Kristy Alaura, Carole Snyder, Scott Disanno, Bob Engelhardt, Shylaja Gattupalli, and their colleagues for organizing, producing, and promoting this project.

As always, I am indebted to my wife, Samantha, for her love, support, and encouragement.

BRIEF CONTENTS

1 Introduction to Computers, Programs, and Java™ 1
2 Elementary Programming 33
3 Selections 77
4 Mathematical Functions, Characters, and Strings 121
5 Loops 159
6 Methods 205
7 Single-Dimensional Arrays 249
8 Multidimensional Arrays 289
9 Objects and Classes 323
10 Object-Oriented Thinking 367
11 Inheritance and Polymorphism 411
12 Exception Handling and Text I/O 453
13 Abstract Classes and Interfaces 499
14 JavaFX Basics 541
15 Event-Driven Programming and Animations 593
16 JavaFX UI Controls and Multimedia 643
17 Binary I/O 691
18 Recursion 719
19 Generics 751
20 Lists, Stacks, Queues, and Priority Queues 775
21 Sets and Maps 815
22 Developing Efficient Algorithms 839
23 Sorting 887
24 Implementing Lists, Stacks, Queues, and Priority Queues 923
25 Binary Search Trees 959
26 AVL Trees 995
27 Hashing 1015
28 Graphs and Applications 1045
29 Weighted Graphs and Applications 1091
30 Aggregate Operations for Collection Streams 1129

CHAPTER 31–44 are available from the Companion Website at www.pearsonhighered.com/liang

31 Advanced JavaFX and FXML
32 Multithreading and Parallel Programming
33 Networking
34 Java Database Programming
35 Advanced Java Database Programming
36 Internationalization
37 Servlets
38 JavaServer Pages
39 JavaServer Faces
40 RMI
41 Web Services
42 2-4 Trees and B-Trees
43 Red-Black Trees
44 Testing Using JUnit

APPENDIXES 1161
A Java Keywords and Reserved Words 1163
B The ASCII Character Set 1164
C Operator Precedence Chart 1166
D Java Modifiers 1168
E Special Floating-Point Values 1170
F Number Systems 1171
G Bitwise Operations 1175
H Regular Expressions 1176
I Enumerated Types 1182
J The Big-O, Big-Omega, and Big-Theta Notations 1187

QUICK REFERENCE 1189
INDEX 1191

CONTENTS

Chapter 1	**Introduction to Computers, Programs, and Java™**	**1**
1.1	Introduction	2
1.2	What Is a Computer?	2
1.3	Programming Languages	7
1.4	Operating Systems	9
1.5	Java, the World Wide Web, and Beyond	10
1.6	The Java Language Specification, API, JDK, JRE, and IDE	11
1.7	A Simple Java Program	12
1.8	Creating, Compiling, and Executing a Java Program	15
1.9	Programming Style and Documentation	18
1.10	Programming Errors	19
1.11	Developing Java Programs Using NetBeans	23
1.12	Developing Java Programs Using Eclipse	26
Chapter 2	**Elementary Programming**	**33**
2.1	Introduction	34
2.2	Writing a Simple Program	34
2.3	Reading Input from the Console	37
2.4	Identifiers	40
2.5	Variables	40
2.6	Assignment Statements and Assignment Expressions	42
2.7	Named Constants	43
2.8	Naming Conventions	44
2.9	Numeric Data Types and Operations	45
2.10	Numeric Literals	48
2.11	JShell	50
2.12	Evaluating Expressions and Operator Precedence	52
2.13	Case Study: Displaying the Current Time	54
2.14	Augmented Assignment Operators	56
2.15	Increment and Decrement Operators	57
2.16	Numeric Type Conversions	58
2.17	Software Development Process	61
2.18	Case Study: Counting Monetary Units	64
2.19	Common Errors and Pitfalls	67
Chapter 3	**Selections**	**77**
3.1	Introduction	78
3.2	`boolean` Data Type, Values, and Expressions	78
3.3	`if` Statements	80
3.4	Two-Way `if-else` Statements	82
3.5	Nested `if` and Multi-Way `if-else` Statements	83
3.6	Common Errors and Pitfalls	85
3.7	Generating Random Numbers	89
3.8	Case Study: Computing Body Mass Index	91
3.9	Case Study: Computing Taxes	92
3.10	Logical Operators	95
3.11	Case Study: Determining Leap Year	99
3.12	Case Study: Lottery	100

3.13	`switch` Statements	102
3.14	Conditional Operators	105
3.15	Operator Precedence and Associativity	106
3.16	Debugging	108

Chapter 4 Mathematical Functions, Characters, and Strings 121

4.1	Introduction	122
4.2	Common Mathematical Functions	122
4.3	Character Data Type and Operations	126
4.4	The String Type	131
4.5	Case Studies	140
4.6	Formatting Console Output	146

Chapter 5 Loops 159

5.1	Introduction	160
5.2	The while Loop	160
5.3	Case Study: Guessing Numbers	163
5.4	Loop Design Strategies	166
5.5	Controlling a Loop with User Confirmation or a Sentinel Value	168
5.6	The `do-while` Loop	171
5.7	The `for` Loop	173
5.8	Which Loop to Use?	176
5.9	Nested Loops	178
5.10	Minimizing Numeric Errors	180
5.11	Case Studies	182
5.12	Keywords *break* and *continue*	186
5.13	Case Study: Checking Palindromes	189
5.14	Case Study: Displaying Prime Numbers	191

Chapter 6 Methods 205

6.1	Introduction	206
6.2	Defining a Method	206
6.3	Calling a Method	208
6.4	`void` vs. Value-Returning Methods	211
6.5	Passing Arguments by Values	213
6.6	Modularizing Code	217
6.7	Case Study: Converting Hexadecimals to Decimals	219
6.8	Overloading Methods	221
6.9	The Scope of Variables	224
6.10	Case Study: Generating Random Characters	225
6.11	Method Abstraction and Stepwise Refinement	227

Chapter 7 Single-Dimensional Arrays 249

7.1	Introduction	250
7.2	Array Basics	250
7.3	Case Study: Analyzing Numbers	257
7.4	Case Study: Deck of Cards	258
7.5	Copying Arrays	260
7.6	Passing Arrays to Methods	261
7.7	Returning an Array from a Method	264
7.8	Case Study: Counting the Occurrences of Each Letter	265
7.9	Variable-Length Argument Lists	268
7.10	Searching Arrays	269

7.11	Sorting Arrays	273
7.12	The `Arrays` Class	274
7.13	Command-Line Arguments	276
Chapter 8	**Multidimensional Arrays**	**289**
8.1	Introduction	290
8.2	Two-Dimensional Array Basics	290
8.3	Processing Two-Dimensional Arrays	293
8.4	Passing Two-Dimensional Arrays to Methods	295
8.5	Case Study: Grading a Multiple-Choice Test	296
8.6	Case Study: Finding the Closest Pair	298
8.7	Case Study: Sudoku	300
8.8	Multidimensional Arrays	303
Chapter 9	**Objects and Classes**	**323**
9.1	Introduction	324
9.2	Defining Classes for Objects	324
9.3	Example: Defining Classes and Creating Objects	326
9.4	Constructing Objects Using Constructors	331
9.5	Accessing Objects via Reference Variables	332
9.6	Using Classes from the Java Library	336
9.7	Static Variables, Constants, and Methods	339
9.8	Visibility Modifiers	344
9.9	Data Field Encapsulation	346
9.10	Passing Objects to Methods	349
9.11	Array of Objects	353
9.12	Immutable Objects and Classes	355
9.13	The Scope of Variables	357
9.14	The `this` Reference	358
Chapter 10	**Object-Oriented Thinking**	**367**
10.1	Introduction	368
10.2	Class Abstraction and Encapsulation	368
10.3	Thinking in Objects	372
10.4	Class Relationships	375
10.5	Case Study: Designing the `Course` Class	378
10.6	Case Study: Designing a Class for Stacks	380
10.7	Processing Primitive Data Type Values as Objects	382
10.8	Automatic Conversion between Primitive Types and Wrapper Class Types	386
10.9	The `BigInteger` and `BigDecimal` Classes	387
10.10	The String Class	388
10.11	The StringBuilder and StringBuffer Classes	395
Chapter 11	**Inheritance and Polymorphism**	**411**
11.1	Introduction	412
11.2	Superclasses and Subclasses	412
11.3	Using the `super` Keyword	418
11.4	Overriding Methods	421
11.5	Overriding vs. Overloading	422
11.6	The `Object` Class and Its `toString()` Method	424
11.7	Polymorphism	425
11.8	Dynamic Binding	425
11.9	Casting Objects and the `instanceof` Operator	429

11.10	The Object's equals Method	433
11.11	The ArrayList Class	434
11.12	Useful Methods for Lists	440
11.13	Case Study: A Custom Stack Class	441
11.14	The `protected` Data and Methods	442
11.15	Preventing Extending and Overriding	445
Chapter 12	**Exception Handling and Text I/O**	**453**
12.1	Introduction	454
12.2	Exception-Handling Overview	454
12.3	Exception Types	459
12.4	Declaring, Throwing, and Catching Exceptions	462
12.5	The `finally` Clause	470
12.6	When to Use Exceptions	472
12.7	Rethrowing Exceptions	472
12.8	Chained Exceptions	473
12.9	Defining Custom Exception Classes	474
12.10	The `File` Class	477
12.11	File Input and Output	480
12.12	Reading Data from the Web	487
12.13	Case Study: Web Crawler	488
Chapter 13	**Abstract Classes and Interfaces**	**499**
13.1	Introduction	500
13.2	Abstract Classes	500
13.3	Case Study: The Abstract `Number` Class	505
13.4	Case Study: `Calendar` and `GregorianCalendar`	507
13.5	Interfaces	510
13.6	The `Comparable` Interface	514
13.7	The `Cloneable` Interface	518
13.8	Interfaces vs. Abstract Classes	523
13.9	Case Study: The `Rational` Class	526
13.10	Class-Design Guidelines	531
Chapter 14	**JavaFX Basics**	**541**
14.1	Introduction	542
14.2	JavaFX vs Swing and AWT	542
14.3	The Basic Structure of a JavaFX Program	542
14.4	Panes, Groups, UI Controls, and Shapes	545
14.5	Property Binding	548
14.6	Common Properties and Methods for Nodes	551
14.7	The `Color` Class	553
14.8	The Font Class	554
14.9	The `Image` and `ImageView` Classes	556
14.10	Layout Panes and Groups	558
14.11	Shapes	567
14.12	Case Study: The `ClockPane` Class	580
Chapter 15	**Event-Driven Programming and Animations**	**593**
15.1	Introduction	594
15.2	Events and Event Sources	596
15.3	Registering Handlers and Handling Events	597
15.4	Inner Classes	601

15.5	Anonymous Inner Class Handlers	602
15.6	Simplifying Event Handling Using Lambda Expressions	605
15.7	Case Study: Loan Calculator	609
15.8	Mouse Events	611
15.9	Key Events	613
15.10	Listeners for Observable Objects	616
15.11	Animation	618
15.12	Case Study: Bouncing Ball	626
15.13	Case Study: US Map	630

Chapter 16 JavaFX UI Controls and Multimedia 643

16.1	Introduction	644
16.2	`Labeled` and `Label`	644
16.3	`Button`	646
16.4	`CheckBox`	648
16.5	`RadioButton`	651
16.6	`TextField`	654
16.7	`TextArea`	655
16.8	`ComboBox`	659
16.9	`ListView`	662
16.10	`ScrollBar`	665
16.11	`Slider`	668
16.12	Case Study: Developing a Tic-Tac-Toe Game	671
16.13	Video and Audio	676
16.14	Case Study: National Flags and Anthems	679

Chapter 17 Binary I/O 691

17.1	Introduction	692
17.2	How Is Text I/O Handled in Java?	692
17.3	Text I/O vs. Binary I/O	693
17.4	Binary I/O Classes	694
17.5	Case Study: Copying Files	704
17.6	Object I/O	706
17.7	Random-Access Files	711

Chapter 18 Recursion 719

18.1	Introduction	720
18.2	Case Study: Computing Factorials	720
18.3	Case Study: Computing Fibonacci Numbers	723
18.4	Problem Solving Using Recursion	726
18.5	Recursive Helper Methods	728
18.6	Case Study: Finding the Directory Size	731
18.7	Case Study: Tower of Hanoi	733
18.8	Case Study: Fractals	736
18.9	Recursion vs. Iteration	740
18.10	Tail Recursion	740

Chapter 19 Generics 751

19.1	Introduction	752
19.2	Motivations and Benefits	752
19.3	Defining Generic Classes and Interfaces	754
19.4	Generic Methods	756
19.5	Case Study: Sorting an Array of Objects	758

19.6	Raw Types and Backward Compatibility	760
19.7	Wildcard Generic Types	761
19.8	Erasure and Restrictions on Generics	764
19.9	Case Study: Generic Matrix Class	766

Chapter 20 Lists, Stacks, Queues, and Priority Queues 775

20.1	Introduction	776
20.2	Collections	776
20.3	Iterators	780
20.4	Using the `forEach` Method	782
20.5	Lists	783
20.6	The `Comparator` Interface	787
20.7	Static Methods for Lists and Collections	792
20.8	Case Study: Bouncing Balls	795
20.9	`Vector` and `Stack` Classes	798
20.10	Queues and Priority Queues	800
20.11	Case Study: Evaluating Expressions	803

Chapter 21 Sets and Maps 815

21.1	Introduction	816
21.2	Sets	816
21.3	Comparing the Performance of Sets and Lists	824
21.4	Case Study: Counting Keywords	827
21.5	Maps	828
21.6	Case Study: Occurrences of Words	833
21.7	Singleton and Unmodifiable Collections and Maps	835

Chapter 22 Developing Efficient Algorithms 839

22.1	Introduction	840
22.2	Measuring Algorithm Efficiency Using Big *O* Notation	840
22.3	Examples: Determining Big *O*	842
22.4	Analyzing Algorithm Time Complexity	846
22.5	Finding Fibonacci Numbers Using Dynamic Programming	849
22.6	Finding Greatest Common Divisors Using Euclid's Algorithm	851
22.7	Efficient Algorithms for Finding Prime Numbers	855
22.8	Finding the Closest Pair of Points Using Divide-and-Conquer	861
22.9	Solving the Eight Queens Problem Using Backtracking	864
22.10	Computational Geometry: Finding a Convex Hull	867
22.11	String Matching	869

Chapter 23 Sorting 887

23.1	Introduction	888
23.2	Insertion Sort	888
23.3	Bubble Sort	890
23.4	Merge Sort	892
23.5	Quick Sort	896
23.6	Heap Sort	900
23.7	Bucket and Radix Sorts	907
23.8	External Sort	909

Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues 923

24.1	Introduction	924
24.2	Common Operations for Lists	924
24.3	Array Lists	928
24.4	Linked Lists	935
24.5	Stacks and Queues	949
24.6	Priority Queues	953

Chapter 25 Binary Search Trees 959

25.1	Introduction	960
25.2	Binary Search Trees Basics	960
25.3	Representing Binary Search Trees	961
25.4	Searching for an Element	962
25.5	Inserting an Element into a BST	962
25.6	Tree Traversal	963
25.7	The BST Class	965
25.8	Deleting Elements from a BST	974
25.9	Tree Visualization and MVC	980
25.10	Iterators	983
25.11	Case Study: Data Compression	985

Chapter 26 AVL Trees 995

26.1	Introduction	996
26.2	Rebalancing Trees	996
26.3	Designing Classes for AVL Trees	999
26.4	Overriding the `insert` Method	1000
26.5	Implementing Rotations	1001
26.6	Implementing the `delete` Method	1002
26.7	The AVLTree Class	1002
26.8	Testing the `AVLTree` Class	1008
26.9	AVL Tree Time Complexity Analysis	1011

Chapter 27 Hashing 1015

27.1	Introduction	1016
27.2	What Is Hashing?	1016
27.3	Hash Functions and Hash Codes	1017
27.4	Handling Collisions Using Open Addressing	1019
27.5	Handling Collisions Using Separate Chaining	1023
27.6	Load Factor and Rehashing	1025
27.7	Implementing a Map Using Hashing	1025
27.8	Implementing Set Using Hashing	1034

Chapter 28 Graphs and Applications 1045

28.1	Introduction	1046
28.2	Basic Graph Terminologies	1047
28.3	Representing Graphs	1048
28.4	Modeling Graphs	1054
28.5	Graph Visualization	1064
28.6	Graph Traversals	1067

28.7 Depth-First Search (DFS) 1068
28.8 Case Study: The Connected Circles Problem 1072
28.9 Breadth-First Search (BFS) 1074
28.10 Case Study: The Nine Tails Problem 1077

Chapter 29 Weighted Graphs and Applications 1091

29.1 Introduction 1092
29.2 Representing Weighted Graphs 1093
29.3 The `WeightedGraph` Class 1095
29.4 Minimum Spanning Trees 1103
29.5 Finding Shortest Paths 1109
29.6 Case Study: The Weighted Nine Tails Problem 1118

Chapter 30 Aggregate Operations for Collection Streams 1129

30.1 Introduction 1130
30.2 Stream Pipelines 1130
30.3 `IntStream`, `LongStream`, and `DoubleStream` 1136
30.4 Parallel Streams 1139
30.5 Stream Reduction Using the `reduce` Method 1141
30.6 Stream Reduction Using the `collect` Method 1144
30.7 Grouping Elements Using the `groupingby` Collector 1147
30.8 Case Studies 1150

Chapter 31–44 are available from the Companion Website at www.pearsonhighered.com/liang

Chapter 31 Advanced JavaFX and FXML

Chapter 32 Multithreading and Parallel Programming

Chapter 33 Networking

Chapter 34 Java Database Programming

Chapter 35 Advanced Database Programming

Chapter 36 Internationalization

Chapter 37 Servlets

Chapter 38 JavaServer Pages

Chapter 39 JavaServer Faces

Chapter 40 RMI

Chapter 41 Web Services

Chapter 42	2-4 Trees and B-Trees	
Chapter 43	Red-Black Trees	
Chapter 44	Testing Using JUnit	
Appendixes		1161
Appendix A	Java Keywords and Reserved Words	1163
Appendix B	The ASCII Character Set	1164
Appendix C	Operator Precedence Chart	1166
Appendix D	Java Modifiers	1168
Appendix E	Special Floating-Point Values	1170
Appendix F	Number Systems	1171
Appendix G	Bitwise Operations	1175
Appendix H	Regular Expressions	1176
Appendix I	Enumerated Types	1182
Appendix J	The Big-O, Big-Omega, and Big-Theta Notations	1187
Quick Reference		1189
Index		1191

VideoNotes

Locations of VideoNotes

http://www.pearsonhighered.com/liang

VideoNote

Chapter 1	Introduction to Computers, Programs, and Java™	1
	Your first Java program	12
	Compile and Run a Java Program	17
	NetBeans brief tutorial	23
	Eclipse brief tutorial	26
Chapter 2	Elementary Programming	33
	Obtain Input	37
	Use operators / and %	54
	Software development process	61
	Compute loan payments	62
	Compute BMI	73
Chapter 3	Selections	77
	Program addition quiz	79
	Program subtraction quiz	89
	Use multi-way `if-else` statements	92
	Sort three integers	112
	Check point location	114
Chapter 4	Mathematical Functions, Characters, and Strings	121
	Introduce Math functions	122
	Introduce strings and objects	131
	Convert hex to decimal	143
	Compute great circle distance	152
	Convert hex to binary	154
Chapter 5	Loops	159
	Use while loop	160
	Guess a number	163
	Multiple subtraction quiz	166
	Use do-while loop	171
	Minimize numeric errors	180
	Display loan schedule	197
	Sum a series	198
Chapter 6	Methods	205
	Define/invoke `max` method	208
	Use `void` method	211
	Modularize code	217
	Stepwise refinement	227
	Reverse an integer	236
	Estimate π	240
Chapter 7	Single-Dimensional Arrays	249
	Random shuffling	254
	Deck of cards	258
	Selection sort	273
	Command-line arguments	277
	Coupon collector's problem	284
	Consecutive four	286
Chapter 8	Multidimensional Arrays	289
	Find the row with the largest sum	294
	Grade multiple-choice test	296
	Sudoku	300
	Multiply two matrices	309
	Even number of 1s	316
Chapter 9	Objects and Classes	323
	Define classes and create objects	324
	Static vs. instance	339
	Data field encapsulation	346
	Immutable objects and this keyword	355
	The this keyword	358
	The Fan class	364
Chapter 10	Object-Oriented Thinking	367
	the Loan class	369
	The `BMI` class	372
	The `StackOfIntegers` class	380
	Process large numbers	387
	The `String` class	388
	The `MyPoint` class	403
Chapter 11	Inheritance and Polymorphism	411
	Geometric class hierarchy	412
	Polymorphism and dynamic binding demo	426
	New `Account` class	448
Chapter 12	Exception Handling and Text I/O	453
	Exception-handling advantages	454
	Create custom exception classes	474
	Write and read data	480
	HexFormatException	493
Chapter 13	Abstract Classes and Interfaces	499
	Abstract GeometricObject class	500
	Calendar and GregorianCalendar classes	507
	The concept of interface	510
Chapter 14	JavaFX Basics	541
	Getting started with JavaFX	542
	Understand property binding	548
	Use `Image` and `ImageView`	556
	Use layout panes	558
	Use shapes	567
	Display a tic-tac-toe board	586
	Display a bar chart	588

Chapter 15	Event-Driven Programming and Animations	593
	Handler and its registration	600
	Anonymous handler	603
	Move message using the mouse	612
	Animate a rising flag	618
	Flashing text	624
	Simple calculator	634
	Check mouse-point location	636
	Display a running fan	639
Chapter 16	JavaFX UI Controls and Multimedia	643
	Use ListView	662
	Use `Slider`	668
	Tic-Tac-Toe	671
	Use Media, MediaPlayer, and MediaView	676
	Use radio buttons and text fields	683
	Set fonts	685
Chapter 17	Binary I/O	691
	Copy file	704
	Object I/O	706
	Split a large file	716
Chapter 18	Recursion	719
	Binary search	730
	Directory size	731
	Search a string in a directory	747
	Recursive tree	750

Animations

Chapter 7	Single-Dimensional Arrays	249
	linear search animation on Companion Website	270
	binary search animation on Companion Website	270
	selection sort animation on Companion Website	273
Chapter 8	Multidimensional Arrays	289
	closest-pair animation on the Companion Website	298
Chapter 22	Developing Efficient Algorithms	839
	binary search animation on the Companion Website	846
	selection sort animation on the Companion Website	846
	closest-pair animation on Companion Website	861
	Eight Queens animation on the Companion Website	864
	convex hull animation on the Companion Website	867
Chapter 23	Sorting	887
	insertion-sort animation on Companion Website	888
	bubble sort animation on the Companion Website	890
	merge animation on Companion Website	894
	partition animation on Companion Website	898
	radix sort on Companion Website	908
Chapter 24	Implementing Lists, Stacks, Queues, and Priority Queues	923
	list animation on Companion Website	924
	stack and queue animation on Companion Website	950
Chapter 25	Binary Search Trees	959
	BST animation on Companion Website	960
Chapter 26	AVL Trees	995
	AVL tree animation on Companion Website	996
Chapter 27	Hashing	1015
	linear probing animation on Companion Website	1020
	quadratic probing animation on Companion Website	1021
	double hashing animation on Companion Website	1022
	separate chaining animation on Companion Website	1025
Chapter 28	Graphs and Applications	1045
	graph learning tool on Companion Website	1048
	U.S. Map Search	1070
Chapter 29	Weighted Graphs and Applications	1091
	weighted graph learning tool on Companion Website	1092

CHAPTER

1

INTRODUCTION TO COMPUTERS, PROGRAMS, AND JAVA™

Objectives

- To understand computer basics, programs, and operating systems (§§1.2–1.4).
- To describe the relationship between Java and the World Wide Web (§1.5).
- To understand the meaning of Java language specification, API, JDK™, JRE™, and IDE (§1.6).
- To write a simple Java program (§1.7).
- To display output on the console (§1.7).
- To explain the basic syntax of a Java program (§1.7).
- To create, compile, and run Java programs (§1.8).
- To use sound Java programming style and document programs properly (§1.9).
- To explain the differences between syntax errors, runtime errors, and logic errors (§1.10).
- To develop Java programs using NetBeans™ (§1.11).
- To develop Java programs using Eclipse™ (§1.12).

1.1 Introduction

Key Point

The central theme of this book is to learn how to solve problems by writing a program.

what is programming?
programming
program

This book is about programming. So, what is programming? The term *programming* means to create (or develop) software, which is also called a *program*. In basic terms, software contains instructions that tell a computer—or a computerized device—what to do.

Software is all around you, even in devices you might not think would need it. Of course, you expect to find and use software on a personal computer, but software also plays a role in running airplanes, cars, cell phones, and even toasters. On a personal computer, you use word processors to write documents, web browsers to explore the Internet, and e-mail programs to send and receive messages. These programs are all examples of software. Software developers create software with the help of powerful tools called *programming languages.*

This book teaches you how to create programs by using the Java programming language. There are many programming languages, some of which are decades old. Each language was invented for a specific purpose—to build on the strengths of a previous language, for example, or to give the programmer a new and unique set of tools. Knowing there are so many programming languages available, it would be natural for you to wonder which one is best. However, in truth, there is no "best" language. Each one has its own strengths and weaknesses. Experienced programmers know one language might work well in some situations, whereas a different language may be more appropriate in other situations. For this reason, seasoned programmers try to master as many different programming languages as they can, giving them access to a vast arsenal of software-development tools.

If you learn to program using one language, you should find it easy to pick up other languages. The key is to learn how to solve problems using a programming approach. That is the main theme of this book.

You are about to begin an exciting journey: learning how to program. At the outset, it is helpful to review computer basics, programs, and operating systems (OSs). If you are already familiar with such terms as central processing unit (CPU), memory, disks, operating systems, and programming languages, you may skip Sections 1.2–1.4.

1.2 What Is a Computer?

Key Point

A computer is an electronic device that stores and processes data.

hardware
software

A computer includes both *hardware* and *software*. In general, hardware comprises the visible, physical elements of the computer, and software provides the invisible instructions that control the hardware and make it perform specific tasks. Knowing computer hardware isn't essential to learning a programming language, but it can help you better understand the effects that a program's instructions have on the computer and its components. This section introduces computer hardware components and their functions.

A computer consists of the following major hardware components (see Figure 1.1):

- A central processing unit (CPU)
- Memory (main memory)
- Storage devices (such as disks and CDs)
- Input devices (such as the mouse and the keyboard)
- Output devices (such as monitors and printers)
- Communication devices (such as modems and network interface cards (NIC))

bus

A computer's components are interconnected by a subsystem called a *bus*. You can think of a bus as a sort of system of roads running among the computer's components; data and power travel along the bus from one part of the computer to another. In personal computers,

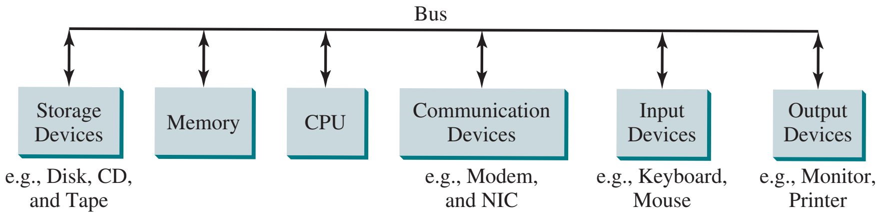


FIGURE 1.1 A computer consists of a CPU, memory, storage devices, input devices, output devices, and communication devices.

the bus is built into the computer's *motherboard*, which is a circuit case that connects all of the parts of a computer together. motherboard

1.2.1 Central Processing Unit

The *central processing unit (CPU)* is the computer's brain. It retrieves instructions from the memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/logic unit*. The control unit controls and coordinates the actions of the other components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, and division) and logical operations (comparisons). CPU

Today's CPUs are built on small silicon semiconductor chips that contain millions of tiny electric switches, called *transistors*, for processing information.

Every computer has an internal clock that emits electronic pulses at a constant rate. These pulses are used to control and synchronize the pace of operations. A higher clock *speed* enables more instructions to be executed in a given period of time. The unit of measurement of clock speed is the *hertz* (*Hz*), with 1 Hz equaling 1 pulse per second. In the 1990s, computers measured clock speed in *megahertz* (*MHz*, i.e., 1 million pulses per second), but CPU speed has been improving continuously; the clock speed of a computer is now usually stated in *gigahertz* (*GHz*, i.e., 1 billion pulses per second). Intel's newest processors run at about 3 GHz. speed hertz megahertz gigahertz

CPUs were originally developed with only one core. The *core* is the part of the processor that performs the reading and executing of instructions. In order to increase the CPU processing power, chip manufacturers are now producing CPUs that contain multiple cores. A multicore CPU is a single component with two or more independent cores. Today's consumer computers typically have two, four, and even eight separate cores. Soon, CPUs with dozens or even hundreds of cores will be affordable. core

1.2.2 Bits and Bytes

Before we discuss memory, let's look at how information (data and programs) are stored in a computer.

A computer is really nothing more than a series of switches. Each switch exists in two states: on or off. Storing information in a computer is simply a matter of setting a sequence of switches on or off. If the switch is on, its value is 1. If the switch is off, its value is 0. These 0s and 1s are interpreted as digits in the binary number system and are called *bits* (binary digits). bits

The minimum storage unit in a computer is a *byte*. A byte is composed of eight bits. A small number such as **3** can be stored as a single byte. To store a number that cannot fit into a single byte, the computer uses several bytes. byte

Data of various kinds, such as numbers and characters, are encoded as a series of bytes. As a programmer, you don't need to worry about the encoding and decoding of data, which the computer system performs automatically, based on the encoding scheme. An *encoding scheme* is a set of rules that govern how a computer translates characters and numbers into data with which the computer can actually work. Most schemes translate each character into encoding scheme a predetermined string of bits. In the popular ASCII encoding scheme, for example, the character **C** is represented as **01000011** in 1 byte.

A computer's storage capacity is measured in bytes and multiples of the byte, as follows:

kilobyte (KB)
megabyte (MB)
gigabyte (GB)
terabyte (TB)

- A *kilobyte (KB)* is about 1,000 bytes.
- A *megabyte (MB)* is about 1 million bytes.
- A *gigabyte (GB)* is about 1 billion bytes.
- A *terabyte (TB)* is about 1 trillion bytes.

A typical one-page word document might take 20 KB. Therefore, 1 MB can store 50 pages of documents, and 1 GB can store 50,000 pages of documents. A typical two-hour high-resolution movie might take 8 GB, so it would require 160 GB to store 20 movies.

1.2.3 Memory

memory

A computer's *memory* consists of an ordered sequence of bytes for storing programs as well as data with which the program is working. You can think of memory as the computer's work area for executing a program. A program and its data must be moved into the computer's memory before they can be executed by the CPU.

unique address
RAM

Every byte in the memory has a *unique address*, as shown in Figure 1.2. The address is used to locate the byte for storing and retrieving the data. Since the bytes in the memory can be accessed in any order, the memory is also referred to as *random-access memory (RAM)*.

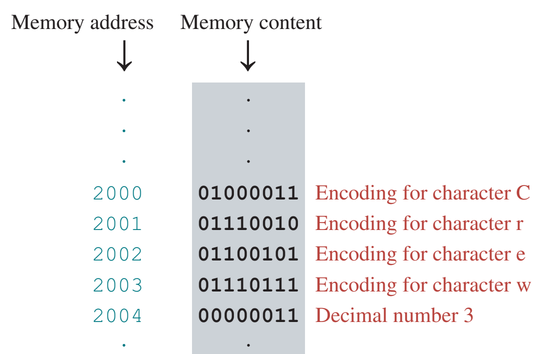


FIGURE 1.2 Memory stores data and program instructions in uniquely addressed memory locations.

Today's personal computers usually have at least 4 GB of RAM, but they more commonly have 8 to 32 GB installed. Generally speaking, the more RAM a computer has, the faster it can operate, but there are limits to this simple rule of thumb.

A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.

Like the CPU, memory is built on silicon semiconductor chips that have millions of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

1.2.4 Storage Devices

storage devices

A computer's memory (RAM) is a volatile form of data storage: Any information that has been saved in memory is lost when the system's power is turned off. Programs and data are permanently stored on *storage devices* and are moved, when the computer actually uses them, to memory, which operates at much faster speeds than permanent storage devices can.

There are four main types of storage devices:

- Magnetic disk drives
- Optical disc drives (CD and DVD)
- Universal serial bus (USB) flash drives
- Cloud storage

Drives are devices for operating a medium, such as disks and CDs. A storage medium physically stores data and program instructions. The drive reads data from the medium and writes data onto the medium. drive

Disks

A computer usually has at least one hard disk drive. *Hard disks* are used for permanently storing data and programs. Newer computers have hard disks that can store from 1 terabyte of data to 4 terabytes of data. Hard disk drives are usually encased inside the computer, but removable hard disks are also available. hard disk

CDs and DVDs

CD stands for compact disc. There are three types of CDs: CD-ROM, CD-R, and CD-RW. A CD-ROM is a prepressed disc. It was popular for distributing software, music, and video. Software, music, and video are now increasingly distributed on the Internet without using CDs. A *CD-R* (CD-Recordable) is a write-once medium. It can be used to record data once and read any number of times. A *CD-RW* (CD-ReWritable) can be used like a hard disk; that is, you can write data onto the disc, then overwrite that data with new data. A single CD can hold up to 700 MB. CD-ROM CD-R CD-RW

DVD stands for digital versatile disc or digital video disc. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a CD; a standard DVD's storage capacity is 4.7 GB. There are two types of DVDs: DVD-R (Recordable) and DVD-RW (ReWritable). DVD

USB Flash Drives

Universal serial bus (USB) connectors allow the user to attach many kinds of peripheral devices to the computer. You can use an USB to connect a printer, digital camera, mouse, external hard disk drive, and other devices to the computer.

An USB *flash drive* is a device for storing and transporting data. A flash drive is small—about the size of a pack of gum. It acts like a portable hard drive that can be plugged into your computer's USB port. USB flash drives are currently available with up to 256 GB storage capacity.

Cloud Storage

Storing data on the cloud is becoming popular. Many companies provide cloud service on the Internet. For example, you can store Microsoft Office documents in Google Docs. Google Docs can be accessed from docs.google.com on the Chrome browser. The documents can be easily shared with others. Microsoft OneDrive is provided free to Windows user for storing files. The data stored in the cloud can be accessed from any device on the Internet.

1.2.5 Input and Output Devices

Input and output devices let the user communicate with the computer. The most common input devices are the *keyboard* and *mouse*. The most common output devices are *monitors* and *printers*.

The Keyboard

A keyboard is a device for entering input. Compact keyboards are available without a numeric keypad.

function key

Function keys are located across the top of the keyboard and are prefaced with the letter *F*. Their functions depend on the software currently being used.

modifier key

A *modifier key* is a special key (such as the *Shift*, *Alt*, and *Ctrl* keys) that modifies the normal action of another key when the two are pressed simultaneously.

numeric keypad

The *numeric keypad*, located on the right side of most keyboards, is a separate set of keys styled like a calculator to use for quickly entering numbers.

arrow keys

Arrow keys, located between the main keypad and the numeric keypad, are used to move the mouse pointer up, down, left, and right on the screen in many kinds of programs.

Insert key
Delete key
Page Up key
Page Down key

The *Insert*, *Delete*, *Page Up*, and *Page Down keys* are used in word processing and other programs for inserting text and objects, deleting text and objects, and moving up or down through a document one screen at a time.

The Mouse

A *mouse* is a pointing device. It is used to move a graphical pointer (usually in the shape of an arrow) called a *cursor* around the screen, or to click on-screen objects (such as a button) to trigger them to perform an action.

The Monitor

The *monitor* displays information (text and graphics). The screen resolution and dot pitch determine the quality of the display.

screen resolution
pixels

The *screen resolution* specifies the number of pixels in horizontal and vertical dimensions of the display device. *Pixels* (short for "picture elements") are tiny dots that form an image on the screen. A common resolution for a 17-inch screen, for example, is 1,024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is.

dot pitch

The *dot pitch* is the amount of space between pixels, measured in millimeters. The smaller the dot pitch, the sharper is the display.

Touchscreens

The cellphones, tablets, appliances, electronic voting machines, as well as some computers use touchscreens. A touchscreen is integrated with a monitor to enable users to enter input and control the display using a finger.

1.2.6 Communication Devices

Computers can be networked through communication devices, such as a dial-up modem (*mo*dulator/*dem*odulator), a digital subscriber line (DSL) or cable modem, a wired network interface card, or a wireless adapter.

dial-up modem

- A *dial-up modem* uses a phone line to dial a phone number to connect to the Internet and can transfer data at a speed up to 56,000 bps (bits per second).

digital subscriber line (DSL)

- A *digital subscriber line (DSL)* connection also uses a standard phone line, but it can transfer data 20 times faster than a standard dial-up modem. Dial-up modem was used in the 90s and is now replaced by DSL and cable modem.

cable modem

- A *cable modem* uses the cable line maintained by the cable company and is generally faster than DSL.

network interface card (NIC)
local area network (LAN)

- A *network interface card (NIC)* is a device that connects a computer to a *local area network (LAN)*. LANs are commonly used to connect computers within a limited area such as a school, a home, and an office. A high-speed NIC called *1000BaseT* can transfer data at 1,000 million bits per second (mbps). million bits per second (mbps)

- Wi-Fi, a special type of wireless networking, is common in homes, businesses, and schools to connect computers, phones, tablets, and printers to the Internet without the need for a physical wired connection. Wi-Fi

Note

Answers to the CheckPoint questions are available at **www.pearsonhighered.com/liang**. Choose this book and click Companion Website to select CheckPoint.

Check Point

1.2.1 What are hardware and software?

1.2.2 List the five major hardware components of a computer.

1.2.3 What does the acronym CPU stand for? What unit is used to measure CPU speed?

1.2.4 What is a bit? What is a byte?

1.2.5 What is memory for? What does RAM stand for? Why is memory called RAM?

1.2.6 What unit is used to measure memory size? What unit is used to measure disk size?

1.2.7 What is the primary difference between memory and a storage device?

1.3 Programming Languages

Computer programs, known as software, are instructions that tell a computer what to do.

Key Point

Computers do not understand human languages, so programs must be written in a language a computer can use. There are hundreds of programming languages, and they were developed to make the programming process easier for people. However, all programs must be converted into the instructions the computer can execute.

1.3.1 Machine Language

A computer's native language, which differs among different types of computers, is its *machine language*—a set of built-in primitive instructions. These instructions are in the form of binary code, so if you want to give a computer an instruction in its native language, you have to enter the instruction as binary code. For example, to add two numbers, you might have to write an instruction in binary code as follows: machine language

```
1101101010011010
```

1.3.2 Assembly Language

Programming in machine language is a tedious process. Moreover, programs written in machine language are very difficult to read and modify. For this reason, *assembly language* was created in the early days of computing as an alternative to machine languages. Assembly language uses a short descriptive word, known as a *mnemonic*, to represent each of the machine-language instructions. For example, the mnemonic **add** typically means to add numbers, and **sub** means to subtract numbers. To add the numbers **2** and **3** and get the result, you might write an instruction in assembly code as follows: assembly language

```
add 2, 3, result
```

Assembly languages were developed to make programming easier. However, because the computer cannot execute assembly language, another program—called an *assembler*—is used to translate assembly-language programs into machine code, as shown in Figure 1.3. assembler

Writing code in assembly language is easier than in machine language. However, it is still tedious to write code in assembly language. An instruction in assembly language essentially

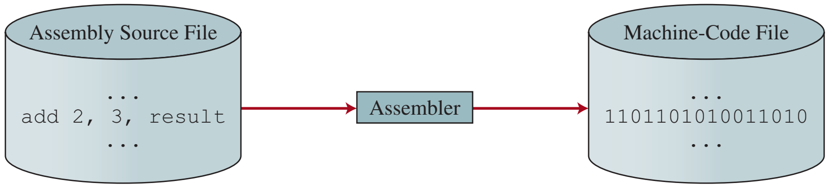


FIGURE 1.3 An assembler translates assembly-language instructions into machine code.

low-level language

corresponds to an instruction in machine code. Writing in assembly language requires that you know how the CPU works. Assembly language is referred to as a *low-level language*, because assembly language is close in nature to machine language and is machine dependent.

1.3.3 High-Level Language

high-level language

statement

In the 1950s, a new generation of programming languages known as *high-level languages* emerged. They are platform independent, which means that you can write a program in a high-level language and run it in different types of machines. High-level languages are similar to English and easy to learn and use. The instructions in a high-level programming language are called *statements*. Here, for example, is a high-level language statement that computes the area of a circle with a radius of **5**:

```
area = 5 * 5 * 3.14159;
```

There are many high-level programming languages, and each was designed for a specific purpose. Table 1.1 lists some popular ones.

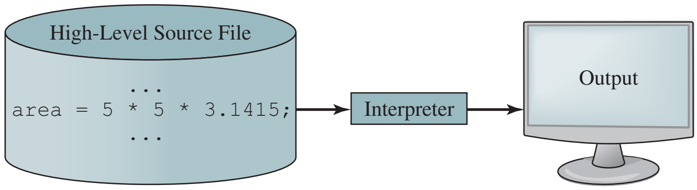
TABLE 1.1 Popular High-Level Programming Languages

Language	*Description*
Ada	Named for Ada Lovelace, who worked on mechanical general-purpose computers. Developed for the Department of Defense and used mainly in defense projects.
BASIC	Beginner's All-purpose Symbolic Instruction Code. Designed to be learned and used easily by beginners.
C	Developed at Bell Laboratories. Combines the power of an assembly language with the ease of use and portability of a high-level language.
C++	An object-oriented language, based on C
C#	Pronounced "C Sharp." An object-oriented programming language developed by Microsoft.
COBOL	COmmon Business Oriented Language. Used for business applications.
FORTRAN	FORmula TRANslation. Popular for scientific and mathematical applications.
Java	Developed by Sun Microsystems, now part of Oracle. An object-oriented programming language, widely used for developing platform-independent Internet applications.
JavaScript	A Web programming language developed by Netscape
Pascal	Named for Blaise Pascal, who pioneered calculating machines in the seventeenth century. A simple, structured, general-purpose language primarily for teaching programming.
Python	A simple general-purpose scripting language good for writing short programs.
Visual Basic	Visual Basic was developed by Microsoft. Enables the programmers to rapidly develop Windows-based applications.

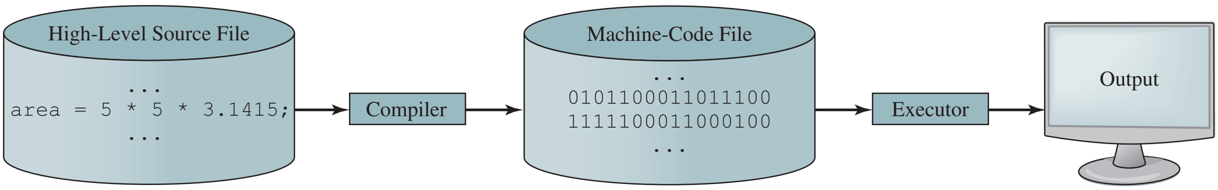
source program
source code
interpreter
compiler

A program written in a high-level language is called a *source program* or *source code*. Because a computer cannot execute a source program, a source program must be translated into machine code for execution. The translation can be done using another programming tool called an *interpreter* or a *compiler*.

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, then executes it right away, as shown in Figure 1.4a. Note a statement from the source code may be translated into several machine instructions.
- A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed, as shown in Figure 1.4b.



(a)



(b)

FIGURE 1.4 (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

1.3.1 What language does the CPU understand?
1.3.2 What is an assembly language? What is an assembler?
1.3.3 What is a high-level programming language? What is a source program?
1.3.4 What is an interpreter? What is a compiler?
1.3.5 What is the difference between an interpreted language and a compiled language?

1.4 Operating Systems

The operating system (OS) *is the most important program that runs on a computer. The OS manages and controls a computer's activities.*

operating system (OS)

The popular *operating systems* for general-purpose computers are Microsoft Windows, Mac OS, and Linux. Application programs, such as a web browser or a word processor, cannot run unless an operating system is installed and running on the computer. Figure 1.5 shows the interrelationship of hardware, operating system, application software, and the user.

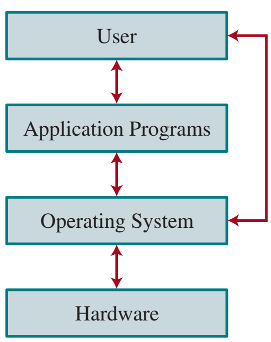


FIGURE 1.5 Users and applications access the computer's hardware via the operating system.

The major tasks of an operating system are as follows:

- Controlling and monitoring system activities
- Allocating and assigning system resources
- Scheduling operations

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and folders on storage devices, and controlling peripheral devices such as disk drives and printers. An operating system must also ensure different programs and users working at the same time do not interfere with each other. In addition, the OS is responsible for security, ensuring unauthorized users and programs are not allowed to access the system.

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (such as CPU time, memory space, disks, and input and output devices) and for allocating and assigning them to run the program.

1.4.3 Scheduling Operations

multiprogramming
multithreading
multiprocessing

The OS is responsible for scheduling programs' activities to make efficient use of system resources. Many of today's operating systems support techniques such as *multiprogramming*, *multithreading*, and *multiprocessing* to increase system performance.

Multiprogramming allows multiple programs such as Microsoft Word, E-mail, and web browser to run simultaneously by sharing the same CPU. The CPU is much faster than the computer's other components. As a result, it is idle most of the time—for example, while waiting for data to be transferred from a disk or waiting for other system resources to respond. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise be idle. For example, multiprogramming enables you to use a word processor to edit a file at the same time as your web browser is downloading a file.

Multithreading allows a single program to execute multiple tasks at the same time. For instance, a word-processing program allows users to simultaneously edit text and save it to a disk. In this example, editing and saving are two tasks within the same program. These two tasks may run concurrently.

Multiprocessing is similar to multithreading. The difference is that multithreading is for running multithreads concurrently within one program, but multiprocessing is for running multiple programs concurrently using multiple processors.

Check Point

1.4.1 What is an operating system? List some popular operating systems.

1.4.2 What are the major responsibilities of an operating system?

1.4.3 What are multiprogramming, multithreading, and multiprocessing?

1.5 Java, the World Wide Web, and Beyond

Key Point

Java is a powerful and versatile programming language for developing software running on mobile devices, desktop computers, and servers.

This book introduces Java programming. Java was developed by a team led by James Gosling at Sun Microsystems. Sun Microsystems was purchased by Oracle in 2010. Originally called *Oak,* Java was designed in 1991 for use in embedded chips in consumer electronic appliances.

In 1995, renamed *Java*, it was redesigned for developing web applications. For the history of Java, see www.java.com/en/javahistory/index.jsp.

Java has become enormously popular. Its rapid rise and wide acceptance can be traced to its design characteristics, particularly its promise that you can write a program once and run it anywhere. As stated by its designer, Java is *simple*, *object oriented*, *distributed*, *interpreted*, *robust*, *secure*, *architecture neutral*, *portable*, *high performance*, *multithreaded*, and *dynamic*. For the anatomy of Java characteristics, see liveexample.pearsoncmg.com/etc/JavaCharacteristics.pdf.

Java is a full-featured, general-purpose programming language that can be used to develop robust mission-critical applications. It is employed not only on desktop computers, but also on servers and mobile devices. Today, more than 3 billion devices run Java. Most major companies use Java in some applications. Most server-side applications were developed using Java. Java was used to develop the code to communicate with and control the robotic rover on Mars. The software for Android cell phones is developed using Java.

Java initially became attractive because Java programs can run from a web browser. Such programs are called *applets*. Today applets are no longer allowed to run from a Web browser due to security issues. Java, however, is now very popular for developing applications on web servers. These applications process data, perform computations, and generate dynamic webpages. Many commercial Websites are developed using Java on the backend.

Check Point

1.5.1 Who invented Java? Which company owns Java now?

1.5.2 What is a Java applet?

1.5.3 What programming language does Android use?

1.6 The Java Language Specification, API, JDK, JRE, and IDE

Key Point

Java syntax is defined in the Java language specification, and the Java library is defined in the Java application program interface (API). The JDK is the software for compiling and running Java programs. An IDE is an integrated development environment for rapidly developing programs.

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will not be able to understand it. The Java language specification and the Java API define the Java standards.

Java language specification

The *Java language specification* is a technical definition of the Java programming language's syntax and semantics. You can find the complete Java language specification at docs.oracle.com/javase/specs/.

API

library

The *application program interface (API)*, also known as *library*, contains predefined classes and interfaces for developing Java programs. The API is still expanding. You can view the latest Java API documentation at https://docs.oracle.com/en/java/javase/11/.

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions:

Java SE, EE, and ME

- *Java Standard Edition (Java SE)* to develop client-side applications. The applications can run on desktop.
- *Java Enterprise Edition (Java EE)* to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
- *Java Micro Edition (Java ME)* to develop applications for mobile devices, such as cell phones.

This book uses Java SE to introduce Java programming. Java SE is the foundation upon which all other Java technology is based. There are many versions of Java SE. The latest, Java Development Toolkit (JDK)

Java SE 11 (or simply Java 11), is used in this book. Oracle releases each version with a *Java Development Toolkit (JDK)*. For Java 11, the Java Development Toolkit is called *JDK 11*.

Java Runtime Environment (JRE)

Integrated development environment

The JDK consists of a set of separate programs, each invoked from a command line, for compiling, running, and testing Java programs. The program for running Java programs is known as *Java Runtime Environment (JRE)*. Instead of using the JDK, you can use a Java development tool (e.g., NetBeans, Eclipse, and TextPad)—software that provides an *integrated development environment (IDE)* for developing Java programs quickly. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. You simply enter source code in one window or open an existing file in a window, and then click a button or menu item or press a function key to compile and run the program.

Check Point

1.6.1 What is the Java language specification?

1.6.2 What does JDK stand for? What does JRE stand for?

1.6.3 What does IDE stand for?

1.6.4 Are tools like NetBeans and Eclipse different languages from Java, or are they dialects or extensions of Java?

1.7 A Simple Java Program

Key Point

A Java program is executed from the **`main`** *method in the class.*

what is a console?

console input

console output

Let's begin with a simple Java program that displays the message **`Welcome to Java!`** on the console. (The word *console* is an old computer term that refers to the text entry and display device of a computer. *Console input* means to receive input from the keyboard, and *console output* means to display output on the monitor.) The program is given in Listing 1.1.

LISTING 1.1 Welcome.java

class

`main` method

display message

VideoNote

Your first Java program

```
1  public class Welcome {
2    public static void main(String[] args) {
3      // Display message Welcome to Java! on the console
4      System.out.println("Welcome to Java!");
5    }
6  }
```

```
Welcome to Java!
```

`line numbers`

Note the *line numbers* are for reference purposes only; they are not part of the program. So, don't type line numbers in your program.

`class name`

Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, *class names* start with an uppercase letter. In this example, the class name is **`Welcome`**.

main method

Line 2 defines the **`main`** method. The program is executed from the **`main`** method. A class may contain several methods. The **`main`** method is the entry point where the program begins execution.

string

statement terminator

A method is a construct that contains statements. The **`main`** method in this program contains the **`System.out.println`** statement. This statement displays the string **`Welcome to Java!`** on the console (line 4). *String* is a programming term meaning a sequence of characters. A string must be enclosed in double quotation marks. Every statement in Java ends with a semicolon (**`;`**), known as the *statement terminator*.

keyword

Keywords have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word **`class`**, it understands that the word after **`class`** is the name for the class. Other keywords in this program are **`public`**, **`static`**, and **`void`**.

comment

Line 3 is a *comment* that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements, and thus are ignored by the compiler. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment*, or enclosed between `/*` and `*/` on one or several lines, called a *block comment* or *paragraph comment*. When the compiler sees `//`, it ignores all text after `//` on the same line. When it sees `/*`, it scans for the next `*/` and ignores any text between `/*` and `*/`. Here are examples of comments:

line comment

block comment

```
// This application program displays Welcome to Java!
/* This application program displays Welcome to Java! */
/* This application program
   displays Welcome to Java! */
```

block

A pair of braces in a program forms a *block* that groups the program's components. In Java, each block begins with an opening brace (**{**) and ends with a closing brace (**}**). Every class has a *class block* that groups the data and methods of the class. Similarly, every method has a *method block* that groups the statements in the method. Blocks can be *nested*, meaning that one block can be placed within another, as shown in the following code:

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

Method block

Class block

Tip

An opening brace must be matched by a closing brace. Whenever you type an opening brace, immediately type a closing brace to prevent the missing-brace error. Most Java IDEs automatically insert the closing brace for each opening brace.

match braces

Caution

Java source programs are case sensitive. It would be wrong, for example, to replace **`main`** in the program with **`Main`**.

case sensitive

special characters

You have seen several special characters (e.g., `{ }`, `//`, `;`) in the program. They are used in almost every program. Table 1.2 summarizes their uses.

common errors

The most common errors you will make as you learn to program will be syntax errors. Like any programming language, Java has its own syntax, and you need to write code that conforms to the *syntax rules*. If your program violates a rule—for example, if the semicolon is missing, a brace is missing, a quotation mark is missing, or a word is misspelled—the Java

syntax rules

TABLE 1.2 Special Characters

Character	*Name*	*Description*
`{}`	Opening and closing braces	Denote a block to enclose statements.
`()`	Opening and closing parentheses	Used with methods.
`[]`	Opening and closing brackets	Denote an array.
`//`	Double slashes	Precede a comment line.
`""`	Opening and closing quotation marks	Enclose a string (i.e., sequence of characters).
`;`	Semicolon	Mark the end of a statement.

compiler will report syntax errors. Try to compile the program with these errors and see what the compiler reports.

> **Note**
> You are probably wondering why the **main** method is defined this way and why **System.out.println(...)** is used to display a message on the console. *For the time being, simply accept that this is how things are done.* Your questions will be fully answered in subsequent chapters.

The program in Listing 1.1 displays one message. Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in Listing 1.2.

Listing 1.2 WelcomeWithThreeMessages.java

class
main method
display message

```java
public class WelcomeWithThreeMessages {
  public static void main(String[] args) {
    System.out.println("Programming is fun!");
    System.out.println("Fundamentals First");
    System.out.println("Problem Driven");
  }
}
```

```
Programming is fun!
Fundamentals First
Problem Driven
```

Further, you can perform mathematical computations and display the result on the console. Listing 1.3 gives an example of evaluating $\frac{10.5 + 2 \times 3}{45 - 3.5}$.

Listing 1.3 ComputeExpression.java

class
main method
compute expression

```java
1 public class ComputeExpression {
2   public static void main(String[] args) {
3     System.out.print("(10.5 + 2 * 3) / (45 - 3.5) = ");
4     System.out.println((10.5 + 2 * 3) / (45 - 3.5));
5   }
6 }
```

```
(10.5 + 2 * 3) / (45 - 3.5) = 0.39759036144578314
```

print vs. println

The **print** method in line 3

```java
System.out.print("(10.5 + 2 * 3) / (45 - 3.5) = ");
```

is identical to the **println** method except that **println** moves to the beginning of the next line after displaying the string, but **print** does not advance to the next line when completed.

The multiplication operator in Java is *****. As you can see, it is a straightforward process to translate an arithmetic expression to a Java expression. We will discuss Java expressions further in Chapter 2.

Check Point

1.7.1 What is a keyword? List some Java keywords.

1.7.2 Is Java case sensitive? What is the case for Java keywords?

1.7.3 What is a comment? Is the comment ignored by the compiler? How do you denote a comment line and a comment paragraph?

1.7.4 What is the statement to display a string on the console?

1.7.5 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    System.out.println("3.5 * 4 / 2 – 2.5 is ");
    System.out.println(3.5 * 4 / 2 – 2.5);
  }
}
```

1.8 Creating, Compiling, and Executing a Java Program

You save a Java program in a .java file and compile it into a .class file. The .class file is executed by the Java Virtual Machine (JVM).

Key Point

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in Figure 1.6. If your program has compile errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

You can use any text editor or IDE to create and edit a Java source-code file. This section demonstrates how to create, compile, and run Java programs from a command window. Sections 1.11 and 1.12 will introduce developing Java programs using NetBeans and Eclipse. From the command window, you can use a text editor such as Notepad to create the Java source-code file, as shown in Figure 1.7.

command window

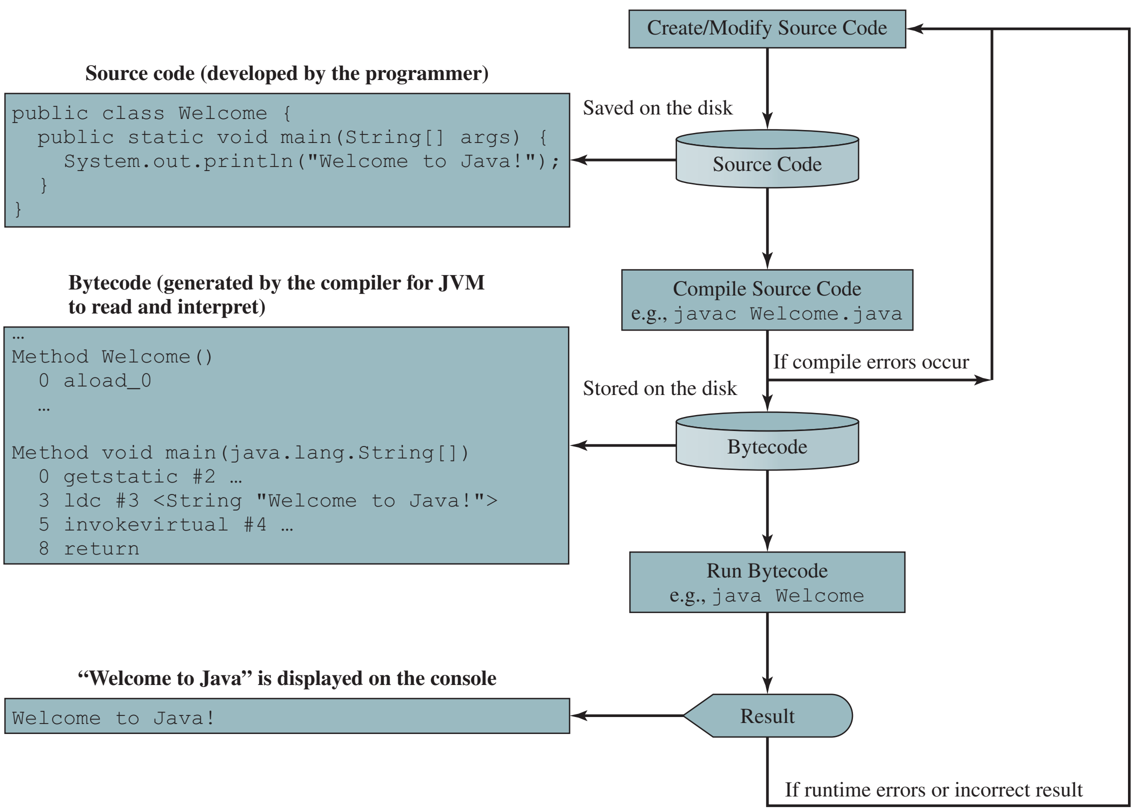


FIGURE 1.6 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.

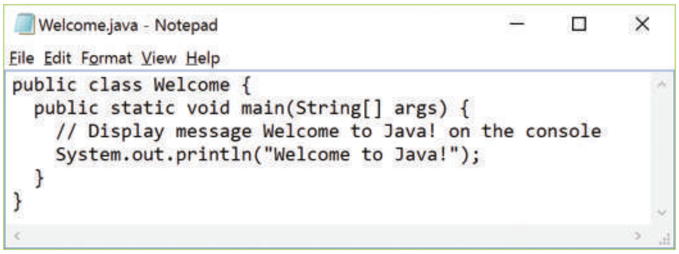


FIGURE 1.7 You can create a Java source file using Windows Notepad.

> **Note**
> The source file must end with the extension `.java` and must have the same exact name as the public class name. For example, the file for the source code in Listing 1.1 should be named **Welcome.java**, since the public class name is `Welcome`.

file name Welcome.java,

compile

A Java compiler translates a Java source file into a Java bytecode file. The following command compiles **Welcome.java**:

```
javac Welcome.java
```

Supplement I.B

Supplement I.C

> **Note**
> You must first install and configure the JDK before you can compile and run programs. See Supplement I.A, Installing and Configuring JDK 11, for how to install the JDK and set up the environment to compile and run Java programs. If you have trouble compiling and running programs, see Supplement I.B, Compiling and Running Java from the Command Window. This supplement also explains how to use basic DOS commands and how to use Windows Notepad to create and edit files. All the supplements are accessible from the Companion Website.

.class bytecode file

bytecode

Java Virtual Machine (JVM)

If there aren't any syntax errors, the *compiler* generates a bytecode file with a `.class` extension. Thus, the preceding command generates a file named **Welcome.class**, as shown in Figure 1.8a. The Java language is a high-level language, but Java bytecode is a low-level language. The *bytecode* is similar to machine instructions but is architecture neutral and can run on any platform that has a *Java Virtual Machine (JVM)*, as shown in Figure 1.8b. Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java's primary advantages: *Java bytecode can run on a variety of hardware platforms and operating systems.* Java source code is compiled into Java bytecode, and Java bytecode is interpreted by the JVM. Your Java code may use the code in the Java library. The JVM executes your code along with the code in the library.

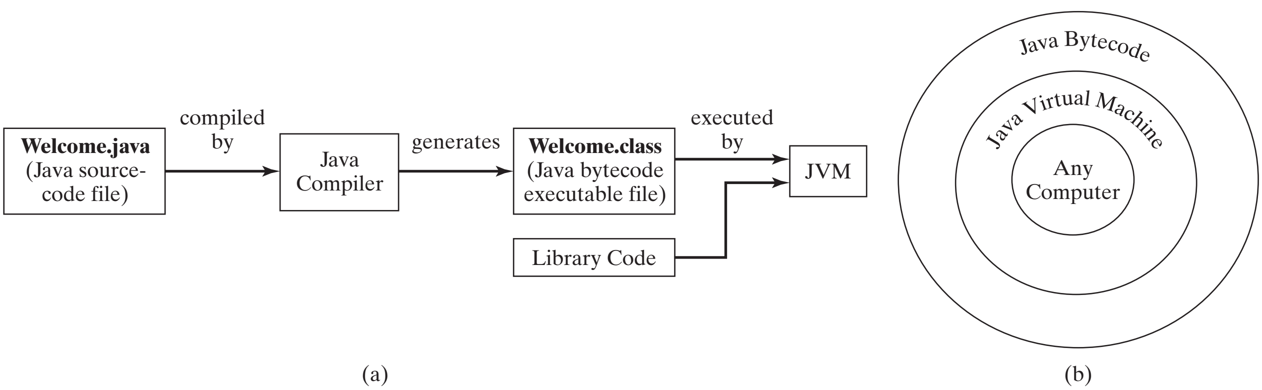


FIGURE 1.8 (a) Java source code is translated into bytecode. (b) Java bytecode can be executed on any computer with a Java Virtual Machine.

To execute a Java program is to run the program's bytecode. You can execute the bytecode on any platform with a JVM, which is an interpreter. It translates the individual instructions in the bytecode into the target machine language code one at a time, rather than the whole program as a single unit. Each step is executed immediately after it is translated.

interpret bytecode

The following command runs the bytecode for Listing 1.1:

run

```
java Welcome
```

Figure 1.9 shows the `javac` command for compiling **Welcome.java**. The compiler generates the **Welcome.class** file, and this file is executed using the `java` command.

`javac` command
`java` command

Note

For simplicity and consistency, all source-code and class files used in this book are placed under **c:\book** unless specified otherwise.

c:\book

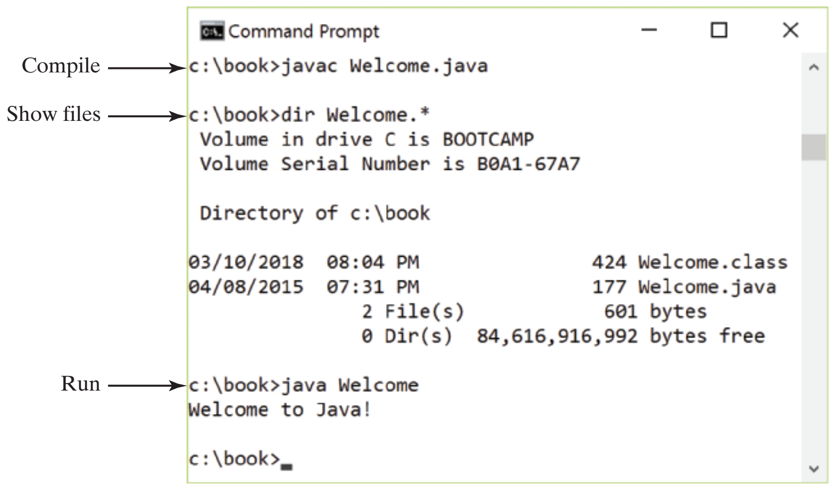


FIGURE 1.9 The output of Listing 1.1 displays the message "Welcome to Java!"

VideoNote
Compile and Run a Java Program

Caution

Do not use the extension `.class` in the command line when executing the program. Use `java ClassName` to run the program. If you use `java ClassName.class` in the command line, the system will attempt to fetch `ClassName.class.class`.

java ClassName

Note

In JDK 11, you can use `java ClassName.java` to compile and run a single-file source code program. This command combines compiling and running in one command. A single-file source code program contains only one class in the file. This is the case for all of our programs in the first eight chapters.

Tip

If you execute a class file that does not exist, a `NoClassDefFoundError` will occur. If you execute a class file that does not have a `main` method or you mistype the `main` method (e.g., by typing `Main` instead of `main`), a `NoSuchMethodError` will occur.

NoClassDefFoundError

NoSuchMethodError

Note

When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the *class loader*. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called the *bytecode verifier* to check the validity of the

class loader

bytecode verifier

bytecode and to ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure Java class files are not tampered with and do not harm your computer.

Pedagogical Note

use package

Your instructor may require you to use packages for organizing programs. For example, you may place all programs in this chapter in a package named *chapter 1*. For instructions on how to use packages, see Supplement I.F, Using Packages to Organize the Classes in the Text.

Check Point

1.8.1 What is the Java source filename extension, and what is the Java bytecode filename extension?

1.8.2 What are the input and output of a Java compiler?

1.8.3 What is the command to compile a Java program?

1.8.4 What is the command to run a Java program?

1.8.5 What is the JVM?

1.8.6 Can Java run on any machine? What is needed to run Java on a computer?

1.8.7 If a `NoClassDefFoundError` occurs when you run a program, what is the cause of the error?

1.8.8 If a `NoSuchMethodError` occurs when you run a program, what is the cause of the error?

1.9 Programming Style and Documentation

Key Point

Good programming style and proper documentation make a program easy to read and help programmers prevent errors.

programming style

documentation

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line, but writing it all on one line would be a bad programming style because it would be hard to read. *Documentation* is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. This section gives several guidelines. For more detailed guidelines, see Supplement I.C, Java Coding Style Guidelines, on the Companion Website.

1.9.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program that explains what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is important to make comments concise so that they do not crowd the program or make it difficult to read.

javadoc comment

In addition to line comments (beginning with `//`) and block comments (beginning with `/*`), Java supports comments of a special type, referred to as *javadoc comments*. javadoc comments begin with `/**` and end with `*/`. They can be extracted into an HTML file using the JDK's **`javadoc`** command. For more information, see Supplement III.X, javadoc Comments, on the Companion Website.

Use javadoc comments (`/** . . . */`) for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted into a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`). To see an example of a javadoc HTML file, check out liveexample.pearsoncmg.com/javadoc/Exercise1.html. Its corresponding Java code is shown in liveexample.pearsoncmg.com/javadoc/Exercise1.txt.

1.9.2 Proper Indentation and Spacing

A consistent indentation style makes programs clear and easy to read, debug, and maintain. *Indentation* is used to illustrate the structural relationships between a program's components or statements. Java can read the program even if all of the statements are on the same long line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least *two* spaces more than the construct within which it is nested.

indent code

A single space should be added on both sides of a binary operator, as shown in (a), rather in (b).

```
System.out.println(3 + 4 * 4);
```

(a) Good style

```
System.out.println(3+4*4);
```

(b) Bad style

1.9.3 Block Styles

A *block* is a group of statements surrounded by braces. There are two popular styles, *next-line* style and *end-of-line* style, as shown below.

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

Next-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

The next-line style aligns braces vertically and makes programs easy to read, whereas the end-of-line style saves space and may help avoid some subtle programming errors. Both are acceptable block styles. The choice depends on personal or organizational preference. You should use a block style consistently—mixing styles is not recommended. This book uses the *end-of-line* style to be consistent with the Java API source code.

Check Point

1.9.1 Reformat the following program according to the programming style and documentation guidelines. Use the end-of-line brace style.

```
public class Test
{
  // Main method
  public static void main(String[] args) {
  /** Display output */
  System.out.println("Welcome to Java");
  }
}
```

1.10 Programming Errors

Key Point

Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

1.10.1 Syntax Errors

Errors that are detected by the compiler are called *syntax errors* or *compile errors*. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace.

syntax errors
compile errors

These errors are usually easy to detect because the compiler tells you where they are and what caused them. For example, the program in Listing 1.4 has a syntax error, as shown in Figure 1.10.

Listing 1.4 ShowSyntaxErrors.java

```
1 public class ShowSyntaxErrors {
2   public static main(String[] args) {
3     System.out.println("Welcome to Java);
4   }
5 }
```

Four errors are reported, but the program actually has two errors:

- The keyword `void` is missing before `main` in line 2.
- The string `Welcome to Java` should be closed with a closing quotation mark in line 3.

Since a single error will often display many lines of compile errors, it is a good practice to fix errors from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

Command Prompt

Compile →

```
c:\book>javac ShowSyntaxErrors.java
ShowSyntaxErrors.java:3: error: invalid method declaration; return type required
  public static main(String[] args) {
                ^
ShowSyntaxErrors.java:4: error: unclosed string literal
    System.out.println("Welcome to Java);
                       ^
ShowSyntaxErrors.java:4: error: ';' expected
    System.out.println("Welcome to Java);
                                         ^
ShowSyntaxErrors.java:6: error: reached end of file while parsing
}
 ^
4 errors

c:\book>
```

Figure 1.10 The compiler reports syntax errors.

fix syntax errors

Tip

If you don't know how to correct an error, compare your program closely, character by character, with similar examples in the text. In the first few weeks of this course, you will probably spend a lot of time fixing syntax errors. Soon you will be familiar with Java syntax, and can quickly fix syntax errors.

1.10.2 Runtime Errors

runtime errors

Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors. An *input error* occurs when the program is waiting for the user to enter a value, but the user enters a value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program.

Another example of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the program in Listing 1.5 would cause a runtime error, as shown in Figure 1.11.

LISTING 1.5 ShowRuntimeErrors.java

```
public class ShowRuntimeErrors {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

runtime error

```
Command Prompt
c:\book>java ShowRuntimeErrors
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ShowRuntimeErrors.main(ShowRuntimeErrors.java:4)

c:\book>
```

Run

FIGURE 1.11 The runtime error causes the program to terminate abnormally.

1.10.3 Logic Errors

logic errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the program in Listing 1.6 to convert Celsius **35** degrees to a Fahrenheit degree:

LISTING 1.6 ShowLogicErrors.java

```
public class ShowLogicErrors {
  public static void main(String[] args) {
    System.out.print("Celsius 35 is Fahrenheit degree ");
    System.out.println((9 / 5) * 35 + 32);
  }
}
```

```
Celsius 35 is Fahrenheit degree 67
```

You will get Fahrenheit **67** degrees, which is wrong. It should be **95.0**. In Java, the division for integers is the quotient—the fractional part is truncated—so in Java **9 / 5** is **1**. To get the correct result, you need to use **9.0 / 5**, which results in **1.8**.

In general, syntax errors are easy to find and easy to correct because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations for the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging. In the upcoming chapters, you will learn the techniques of tracing programs and finding logic errors.

1.10.4 Common Errors

Missing a closing brace, missing a semicolon, missing quotation marks for strings, and misspelling names are common errors for new programmers.

Common Error 1: Missing Braces

The braces are used to denote a block in the program. Each opening brace must be matched by a closing brace. A common error is missing the closing brace. To avoid this error, type a closing brace whenever an opening brace is typed, as shown in the following example:

```
public class Welcome {

}
```

← Type this closing brace right away to match the opening brace

If you use an IDE such as NetBeans and Eclipse, the IDE automatically inserts a closing brace for each opening brace typed.

Common Error 2: Missing Semicolons

Each statement ends with a statement terminator (;). Often, a new programmer forgets to place a statement terminator for the last statement in a block, as shown in the following example:

```
public static void main(String[] args) {
  System.out.println("Programming is fun!");
  System.out.println("Fundamentals First");
  System.out.println("Problem Driven")
}
```

Missing a semicolon

Common Error 3: Missing Quotation Marks

A string must be placed inside the quotation marks. Often, a new programmer forgets to place a quotation mark at the end of a string, as shown in the following example:

```
System.out.println("Problem Driven );
```

Missing a quotation mark

If you use an IDE such as NetBeans and Eclipse, the IDE automatically inserts a closing quotation mark for each opening quotation mark typed.

Common Error 4: Misspelling Names

Java is case sensitive. Misspelling names is a common error for new programmers. For example, the word `main` is misspelled as `Main` and `String` is misspelled as `string` in the following code:

```
public class Test {
  public static void Main(string[] args) {
    System.out.println((10.5 + 2 * 3) / (45 - 3.5));
  }
}
```

Check Point

1.10.1 What are syntax errors (compile errors), runtime errors, and logic errors?

1.10.2 Give examples of syntax errors, runtime errors, and logic errors.

1.10.3 If you forget to put a closing quotation mark on a string, what kind of error will be raised?

1.10.4 If your program needs to read integers, but the user entered strings, an error would occur when running this program. What kind of error is this?

1.10.5 Suppose you write a program for computing the perimeter of a rectangle and you mistakenly write your program so it computes the area of a rectangle. What kind of error is this?

1.10.6 Identify and fix the errors in the following code:

```
public class Welcome {
  public void Main(String[] args) {
    System.out.println('Welcome to Java!);
  }
)
```

1.11 Developing Java Programs Using NetBeans

You can edit, compile, run, and debug Java Programs using NetBeans.

Key Point

> **Note**
> Section 1.8 introduced developing programs from the command line. Many of our readers also use an IDE. This section and next section introduce two most popular Java IDEs: NetBeans and Eclipse. These two sections may be skipped.

NetBeans and Eclipse are two free popular integrated development environments for developing Java programs. They are easy to learn if you follow simple instructions. We recommend that you use either one for developing Java programs. This section gives the essential instructions to guide new users to create a project, create a class, compile, and run a class in NetBeans. The use of Eclipse will be introduced in the next section. To use JDK 11, you need NetBeans 9 or higher. For instructions on downloading and installing latest version of NetBeans, see Supplement II.B.

VideoNote
NetBeans brief tutorial

1.11.1 Creating a Java Project

Before you can create Java programs, you need to first create a project. A project is like a folder to hold Java programs and all supporting files. You need to create a project only once. Here are the steps to create a Java project:

1. Choose *File*, *New Project* to display the New Project dialog box, as shown in Figure 1.12.
2. Select Java in the Categories section and Java Application in the Projects section, and then click *Next* to display the New Java Application dialog box, as shown in Figure 1.13.
3. Type `demo` in the Project Name field and `c:\michael` in Project Location field. Uncheck *Use Dedicated Folder for Storing Libraries* and uncheck *Create Main Class*.
4. Click *Finish* to create the project, as shown in Figure 1.14.

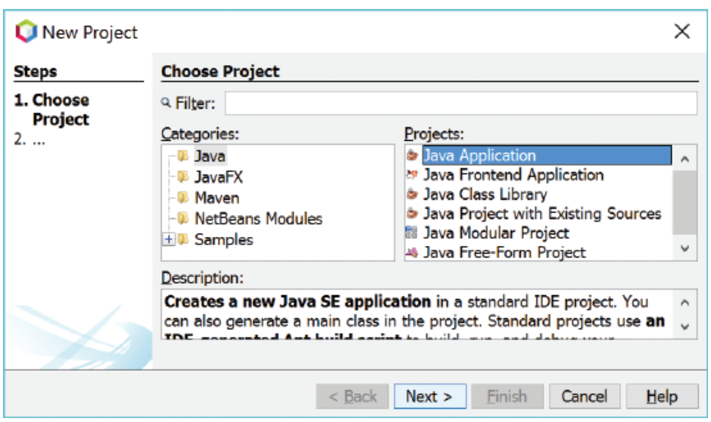


FIGURE 1.12 The New Project dialog is used to create a new project and specify a project type.

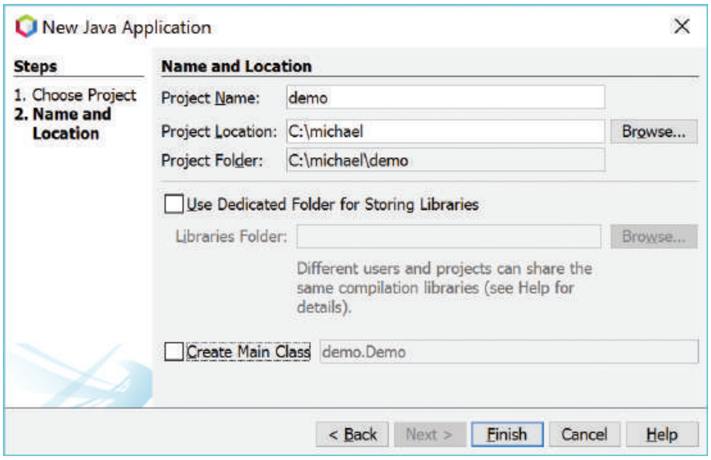


Figure 1.13 The New Java Application dialog is for specifying a project name and location. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

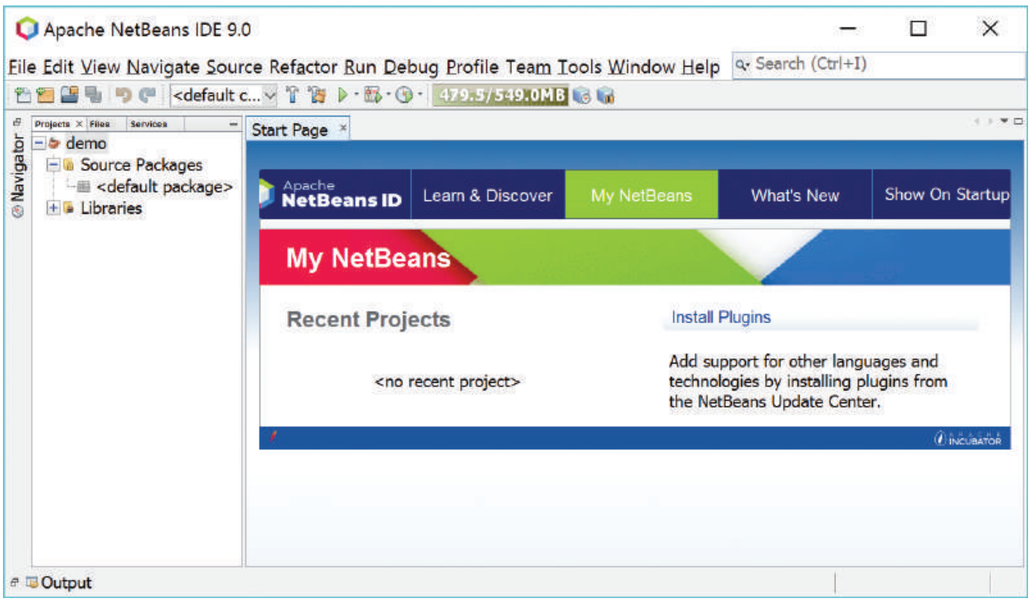


Figure 1.14 A New Java project named demo is created. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

1.11.2 Creating a Java Class

After a project is created, you can create Java programs in the project using the following steps:

1. Right-click the demo node in the project pane to display a context menu. Choose *New*, *Java Class* to display the New Java Class dialog box, as shown in Figure 1.15.

2. Type `Welcome` in the Class Name field and select the Source Packages in the Location field. Leave the Package field blank. This will create a class in the default package.

3. Click *Finish* to create the Welcome class. The source-code file **Welcome.java** is placed under the <default package> node.
4. Modify the code in the Welcome class to match Listing 1.1 in the text, as shown in Figure 1.16.

1.11.3 Compiling and Running a Class

To run **Welcome.java**, right-click **Welcome.java** to display a context menu and choose *Run File*, or simply press Shift + F6. The output is displayed in the Output pane, as shown in Figure 1.16. The *Run File* command automatically compiles the program if the program has been changed.

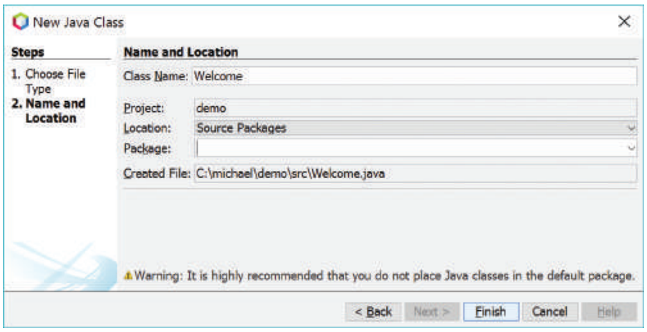


FIGURE 1.15 The New Java Class dialog box is used to create a new Java class. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

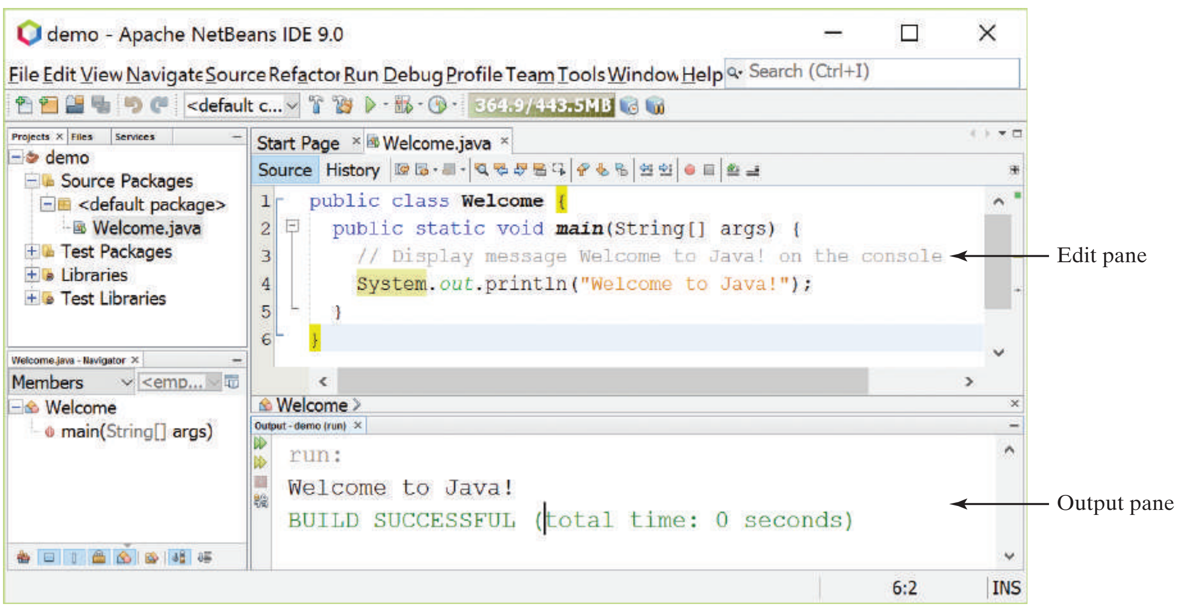


FIGURE 1.16 You can edit a program and run it in NetBeans. *Source*: Copyright © 1995–2016 Oracle and/or its affiliates. All rights reserved. Used with permission.

1.12 Developing Java Programs Using Eclipse

You can edit, compile, run, and debug Java Programs using Eclipse.

Key Point

The preceding section introduced developing Java programs using NetBeans. You can also use Eclipse to develop Java programs. This section gives the essential instructions to guide new users to create a project, create a class, and compile/run a class in Eclipse. To use JDK 11, you need Eclipse 4.9 or higher. For instructions on downloading and installing latest version of Eclipse, see Supplement II.D.

1.12.1 Creating a Java Project

VideoNote

Eclipse brief tutorial

Before creating Java programs in Eclipse, you need to first create a project to hold all files. Here are the steps to create a Java project in Eclipse:

1. Choose *File*, *New*, *Java Project* to display the New Project wizard, as shown in Figure 1.17.

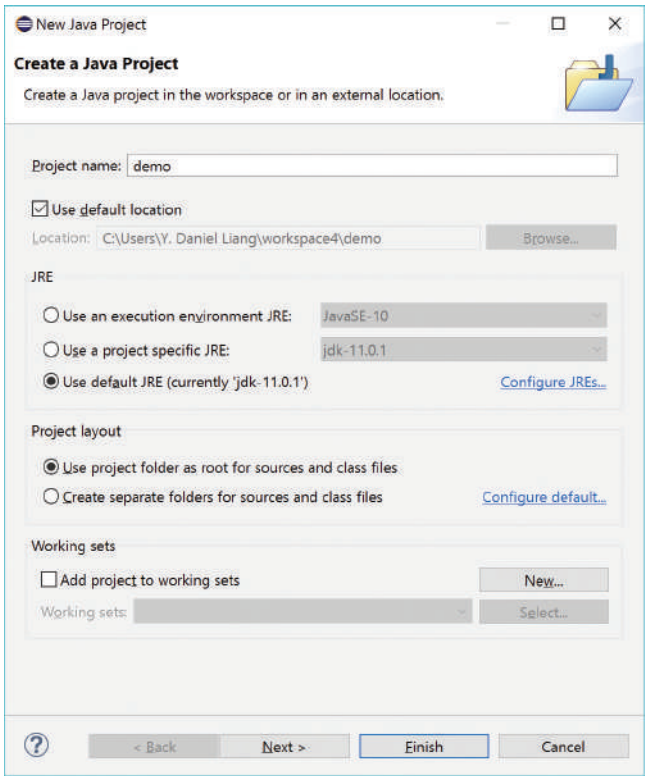


FIGURE 1.17 The New Java Project dialog is for specifying a project name and the properties.
Source: Eclipse Foundation, Inc.

2. Type **demo** in the Project name field. As you type, the Location field is automatically set by default. You may customize the location for your project.
3. Make sure you selected the options *Use project folder as root for sources and class files* so the .java and .class files are in the same folder for easy access.
4. Click *Finish* to create the project, as shown in Figure 1.18.

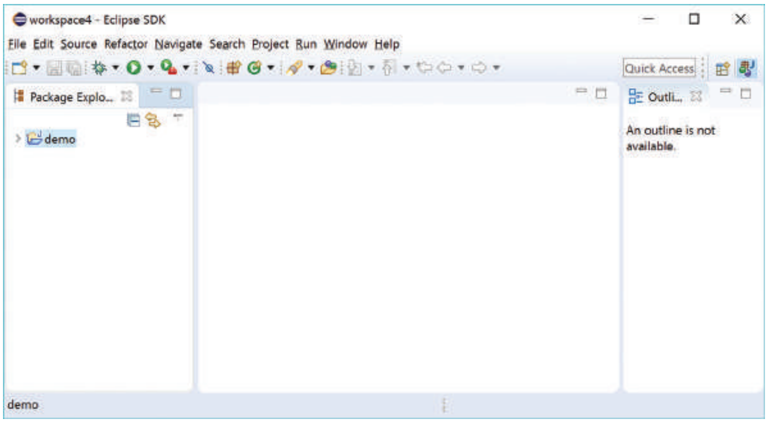


FIGURE 1.18 A New Java project named demo is created. *Source*: Eclipse Foundation, Inc.

1.12.2 Creating a Java Class

After a project is created, you can create Java programs in the project using the following steps:

1. Choose *File*, *New*, *Class* to display the New Java Class wizard.
2. Type `Welcome` in the Name field.
3. Check the option *public static void main(String[] args)*.
4. Click *Finish* to generate the template for the source code **Welcome.java**, as shown in Figure 1.19.

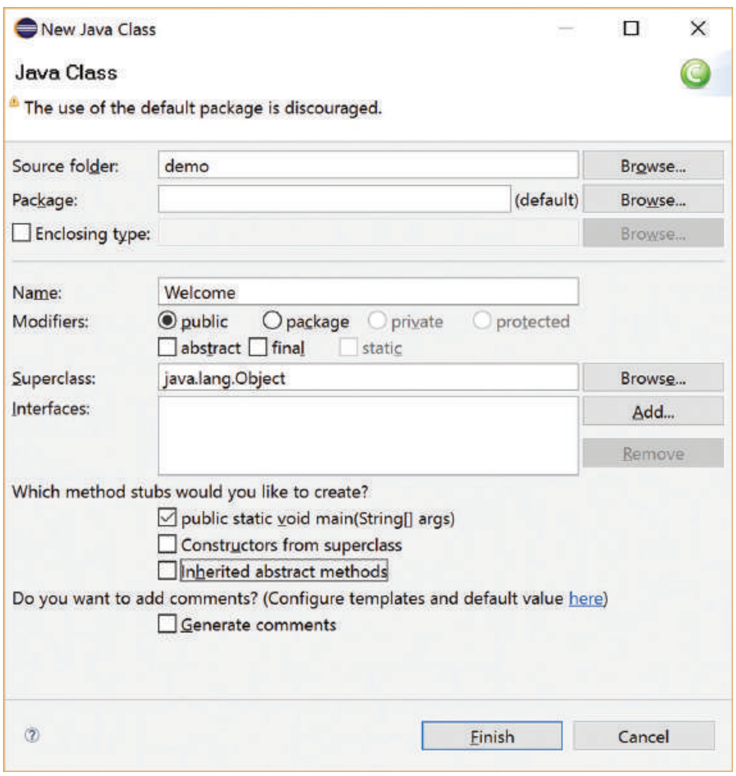


FIGURE 1.19 The New Java Class dialog box is used to create a new Java class. *Source*: Eclipse Foundation, Inc.

1.12.3 Compiling and Running a Class

To run the program, right-click the class in the project to display a context menu. Choose *Run*, *Java Application* in the context menu to run the class. The output is displayed in the Console pane, as shown in Figure 1.20. The *Run* command automatically compiles the program if the program has been changed.

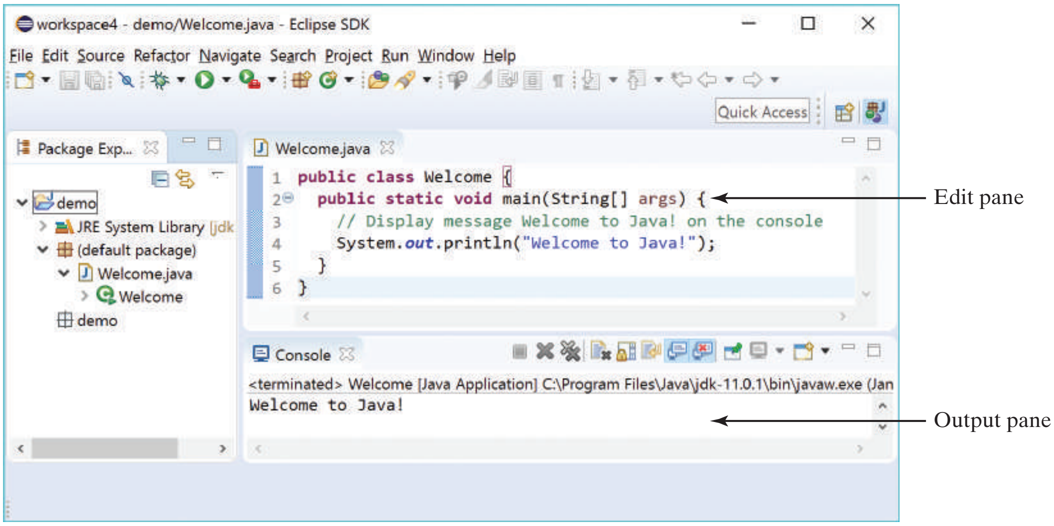


Figure 1.20 You can edit a program and run it in Eclipse. *Source*: Eclipse Foundation, Inc.

Key Terms

Application Program Interface (API) 11
assembler 7
assembly language 7
bit 3
block 13
block comment 13
bus 2
byte 3
bytecode 16
bytecode verifier 17
cable modem 6
central processing unit (CPU) 3
class loader 17
comment 13
compiler 8
console 12
dial-up modem 6
dot pitch 6
DSL (digital subscriber line) 6
encoding scheme 3
hardware 2
high-level language 8
integrated development environment (IDE) 12
interpreter 8
java *command* 17
Java Development Toolkit (JDK) 12
Java language specification 11
Java Runtime Environment (JRE) 12
Java Virtual Machine (JVM) 16
javac *command* 17
keyword 12
library 11
line comment 13
logic error 21
low-level language 8
machine language 7
main *method* 12
memory 4
motherboard 3
network interface card (NIC) 6
operating system (OS) 9
pixel 6
program 2
programming 2
runtime error 20
screen resolution 6
software 2

source code 8
source program 8
statement 8
statement terminator 12
storage devices 4
syntax error 19

Note
The above terms are defined in this chapter. Glossary (at the end of TOC) lists all the key terms and descriptions in the book, organized by chapters.

Supplement I.A

Chapter Summary

1. A computer is an electronic device that stores and processes data.
2. A computer includes both *hardware* and *software.*
3. Hardware is the physical aspect of the computer that can be touched.
4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.
5. Computer *programming* is the writing of instructions (i.e., code) for computers to perform.
6. The *central processing unit (CPU)* is a computer's brain. It retrieves instructions from *memory* and executes them.
7. Computers use zeros and ones because digital devices have two stable states, referred to by convention as zero and one.
8. A *bit* is a binary digit 0 or 1.
9. A *byte* is a sequence of 8 bits.
10. A kilobyte is about 1,000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1,000 gigabytes.
11. Memory stores data and program instructions for the CPU to execute.
12. A memory unit is an ordered sequence of bytes.
13. Memory is volatile, because information is lost when the power is turned off.
14. Programs and data are permanently stored on *storage devices* and are moved to memory when the computer actually uses them.
15. The *machine language* is a set of primitive instructions built into every computer.
16. *Assembly language* is a *low-level programming language* in which a mnemonic is used to represent each machine-language instruction.
17. *High-level languages* are English-like and easy to learn and program.
18. A program written in a high-level language is called a *source program.*
19. A *compiler* is a software program that translates the source program into a *machine-language program.*
20. The *operating system (OS)* is a program that manages and controls a computer's activities.

21. Java is platform independent, meaning you can write a program once and run it on any computer.
22. The Java source file name must match the public class name in the program. Java source-code files must end with the `.java` extension.
23. Every class is compiled into a separate bytecode file that has the same name as the class and ends with the `.class` extension.
24. To compile a Java source-code file from the command line, use the `javac` command.
25. To run a Java class from the command line, use the `java` command.
26. Every Java program is a set of class definitions. The keyword `class` introduces a class definition. The contents of the class are included in a *block*.
27. A block begins with an opening brace (`{`) and ends with a closing brace (`}`).
28. Methods are contained in a class. To run a Java program, the program must have a `main` method. The `main` method is the entry point where the program starts when it is executed.
29. Every *statement* in Java ends with a semicolon (`;`), known as the *statement terminator*.
30. *Keywords* have a specific meaning to the compiler and cannot be used for other purposes in the program.
31. In Java, comments are preceded by two slashes (`//`) on a line, called a *line comment*, or enclosed between `/*` and `*/` on one or several lines, called a *block comment* or *paragraph comment*. Comments are ignored by the compiler.
32. Java source programs are case sensitive.
33. Programming errors can be categorized into three types: *syntax errors*, *runtime errors*, and *logic errors*. Errors reported by a compiler are called syntax errors or *compile errors*. Runtime errors are errors that cause a program to terminate abnormally. Logic errors occur when a program does not perform the way it was intended to.

Quiz

Answer the quiz for this chapter at www.pearsonhighered.com/liang. Choose this book and click Companion Website to select Quiz.

MyProgrammingLab™

Programming Exercises

Pedagogical Note

We cannot stress enough the importance of learning programming through exercises. For this reason, the book provides a large number of programming exercises at various levels of difficulty. The problems cover many application areas, including math, science, business, financial, gaming, animation, and multimedia. Solutions to most even-numbered programming exercises are on the Companion Website. Solutions to most odd-numbered programming exercises are on the Instructor Resource Website. The level of difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***).

level of difficulty

1.1 (*Display three messages*) Write a program that displays `Welcome to Java`, `Welcome to Computer Science`, and `Programming is fun`.

1.2 (*Display five messages*) Write a program that displays `Welcome to Java` five times.

***1.3** (*Display a pattern*) Write a program that displays the following pattern:

```
    J     A     V     V     A
    J    A A     V   V     A A
J   J   AAAAA     V V     AAAAA
 J J   A     A     V     A     A
```

1.4 (*Print a table*) Write a program that displays the following table:

```
a      a^2    a^3
1      1      1
2      4      8
3      9      27
4      16     64
```

1.5 (*Compute expressions*) Write a program that displays the result of

$$\frac{9.5 \times 4.5 - 2.5 \times 3}{45.5 - 3.5}.$$

1.6 (*Summation of a series*) Write a program that displays the result of

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9.$$

1.7 (*Approximate π*) π can be computed using the following formula:

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots\right)$$

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}\right)$ and $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13}\right)$. Use `1.0` instead of `1` in your program.

1.8 (*Area and perimeter of a circle*) Write a program that displays the area and perimeter of a circle that has a radius of `5.5` using the following formulas:

$$perimeter = 2 \times radius \times \pi$$

$$area = radius \times radius \times \pi$$

1.9 (*Area and perimeter of a rectangle*) Write a program that displays the area and perimeter of a rectangle with a width of `4.5` and a height of `7.9` using the following formula:

$$area = width \times height$$

1.10 (*Average speed in miles*) Assume that a runner runs `14` kilometers in `45` minutes and `30` seconds. Write a program that displays the average speed in miles per hour. (Note `1` mile is equal to `1.6` kilometers.)

***1.11** (*Population projection*) The U.S. Census Bureau projects population based on the following assumptions:

- One birth every 7 seconds
- One death every 13 seconds
- One new immigrant every 45 seconds

Write a program to display the population for each of the next five years. Assume that the current population is 312,032,486, and one year has 365 days. *Hint*: In Java, if two integers perform division, the result is an integer. The fractional part is truncated. For example, `5 / 4` is `1` (not `1.25`) and `10 / 4` is `2` (not `2.5`). To get an accurate result with the fractional part, one of the values involved in the division must be a number with a decimal point. For example, `5.0 / 4` is `1.25` and `10 / 4.0` is `2.5`.

1.12 (*Average speed in kilometers*) Assume that a runner runs `24` miles in `1` hour, `40` minutes, and `35` seconds. Write a program that displays the average speed in kilometers per hour. (Note `1` mile is equal to `1.6` kilometers.)

***1.13** (*Algebra: solve* 2×2 *linear equations*) You can use Cramer's rule to solve the following 2×2 system of linear equation provided that $ad - bc$ is not 0:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \qquad x = \frac{ed - bf}{ad - bc} \qquad y = \frac{af - ec}{ad - bc}$$

Write a program that solves the following equation and displays the value for x and y: (Hint: replace the symbols in the formula with numbers to compute x and y. This exercise can be done in Chapter 1 without using materials in later chapters.)

$$\begin{aligned} 3.4x + 50.2y &= 44.5 \\ 2.1x + .55y &= 5.9 \end{aligned}$$

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER 2

ELEMENTARY PROGRAMMING

Objectives

- To write Java programs to perform simple computations (§2.2).
- To obtain input from the console using the `Scanner` class (§2.3).
- To use identifiers to name variables, constants, methods, and classes (§2.4).
- To use variables to store data (§§2.5 and 2.6).
- To program with assignment statements and assignment expressions (§2.6).
- To use constants to store permanent data (§2.7).
- To name classes, methods, variables, and constants by following their naming conventions (§2.8).
- To explore Java numeric primitive data types: `byte`, `short`, `int`, `long`, `float`, and `double` (§2.9).
- To read a `byte`, `short`, `int`, `long`, `float`, or `double` value from the keyboard (§2.9.1).
- To perform operations using operators `+`, `-`, `*`, `/`, and `%` (§2.9.2).
- To perform exponent operations using `Math.pow(a, b)` (§2.9.3).
- To write integer literals, floating-point literals, and literals in scientific notation (§2.10).
- To use JShell to quickly test Java code (§2.11).
- To write and evaluate numeric expressions (§2.12).
- To obtain the current system time using `System.currentTimeMillis()` (§2.13).
- To use augmented assignment operators (§2.14).
- To distinguish between postincrement and preincrement and between postdecrement and predecrement (§2.15).
- To cast the value of one type to another type (§2.16).
- To describe the software development process and apply it to develop the loan payment program (§2.17).
- To write a program that converts a large amount of money into smaller units (§2.18).
- To avoid common errors and pitfalls in elementary programming (§2.19).

2.1 Introduction

Key Point

The focus of this chapter is on learning elementary programming techniques to solve problems.

In Chapter 1, you learned how to create, compile, and run very basic Java programs. You will learn how to solve problems by writing programs. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

Suppose, for example, you need to take out a student loan. Given the loan amount, loan term, and annual interest rate, can you write a program to compute the monthly payment and total payment? This chapter shows you how to write programs like this. Along the way, you will learn the basic steps that go into analyzing a problem, designing a solution, and implementing the solution by creating a program.

2.2 Writing a Simple Program

Key Point

Writing a program involves designing a strategy for solving the problem then using a programming language to implement that strategy.

problem

Let's first consider the simple *problem* of computing the area of a circle. How do we write a program for solving this problem?

algorithm

pseudocode

Writing a program involves designing algorithms and translating algorithms into programming instructions, or code. An *algorithm* lists the steps you can follow to solve a problem. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code). The algorithm for calculating the area of a circle can be described as follows:

1. Read in the circle's radius.
2. Compute the area using the following formula:

$$area = radius \times radius \times \pi$$

3. Display the result.

Tip

It's always a good practice to outline your program (or its underlying problem) in the form of an algorithm before you begin coding.

When you *code*—that is, when you write a program—you translate an algorithm into a program. You already know every Java program begins with a class definition in which the keyword **`class`** is followed by the class name. Assume you have chosen **`ComputeArea`** as the class name. The outline of the program would look as follows:

```
public class ComputeArea {
  // Details to be given later
}
```

As you know, every Java program must have a **`main`** method where program execution begins. The program is then expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    // Step 1: Read in radius

    // Step 2: Compute area
```

```
    // Step 3: Display the area
  }
}
```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius
- Storing the radius in the program

variable

Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable*. A variable represents a value stored in the computer's memory.

descriptive names

data type

declare variables

primitive data types

Rather than using `x` and `y` as variable names, choose *descriptive names*: in this case, `radius` for radius and `area` for area. To let the compiler know what `radius` and `area` are, specify their *data types*. That is the kind of data stored in a variable, whether an integer, real number, or something else. This is known as *declaring variables*. Java provides simple data types for representing integers, real numbers, characters, and Boolean types. These types are known as *primitive data types* or *fundamental types.*

floating-point numbers

Real numbers (i.e., numbers with a decimal point) are represented using a method known as *floating-point* in computers. Therefore, the real numbers are also called *floating-point numbers*. In Java, you can use the keyword `double` to declare a floating-point variable. Declare `radius` and `area` as `double`. The program can be expanded as follows:

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius;
    double area;

    // Step 1: Read in radius

    // Step 2: Compute area

    // Step 3: Display the area
  }
}
```

The program declares `radius` and `area` as variables. The keyword `double` indicates that `radius` and `area` are floating-point values stored in the computer.

The first step is to prompt the user to designate the circle's `radius`. You will soon learn how to prompt the user for information. For now, to learn how variables work, you can assign a fixed value to `radius` in the program as you write the code. Later, you'll modify the program to prompt the user for this value.

The second step is to compute `area` by assigning the result of the expression `radius * radius * 3.14159` to `area`.

In the final step, the program will display the value of `area` on the console by using the `System.out.println` method.

Listing 2.1 shows the complete program, and a sample run of the program is shown in Figure 2.1.

LISTING 2.1 ComputeArea.java

```
public class ComputeArea {
  public static void main(String[] args) {
    double radius; // Declare radius
    double area; // Declare area

    // Assign a radius
    radius = 20; // radius is now 20
```

```
 8
 9      // Compute area
10      area = radius * radius * 3.14159;
11
12      // Display results
13      System.out.println("The area for the circle of radius " +
14        radius + " is " + area);
15    }
16  }
```

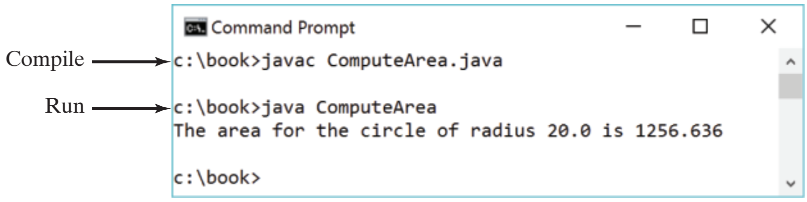


FIGURE 2.1 The program displays the area of a circle.

declare variable

assign value

tracing program

Variables such as **radius** and **area** correspond to memory locations. Every variable has a name, a type, and a value. Line 3 declares that **radius** can store a **double** value. The value is not defined until you assign a value. Line 7 assigns **20** into the variable **radius**. Similarly, line 4 declares the variable **area**, and line 10 assigns a value into **area**. The following table shows the value in the memory for **area** and **radius** as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. This method of reviewing how a program works is called *tracing a program*. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.

line#	radius	area
3	no value	
4		no value
7	20	
10		1256.636

concatenate strings

concatenate strings with numbers

The plus sign (**+**) has two meanings: one for addition, and the other for concatenating (combining) strings. The plus sign (**+**) in lines 13–14 is called a *string concatenation operator*. It combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string. Therefore, the plus signs (**+**) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in Chapter 4.

Caution

A string cannot cross lines in the source code. Thus, the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,
by Y. Daniel Liang");
```

break a long string

To fix the error, break the string into separate substrings, and use the concatenation operator (**+**) to combine them:

```
System.out.println("Introduction to Java Programming, " +
  "by Y. Daniel Liang");
```

2.2.1 Identify and fix the errors in the following code:

Check Point

```
public class Test {
  public void main(string[] args) {
    double i = 50.0;
    double k = i + 50.0;
    double j = k + 1;

    System.out.println("j is " + j + " and
      k is " + k);
  }
}
```

2.3 Reading Input from the Console

Reading input from the console enables the program to accept input from the user.

Key Point

In Listing 2.1, the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient, so instead you can use the **Scanner** class for console input.

VideoNote
Obtain Input

Java uses **System.out** to refer to the standard output device, and **System.in** to the standard input device. By default, the output device is the display monitor, and the input device is the keyboard. To perform console output, you simply use the **println** method to display a primitive value or a string to the console. To perform console input, you need to use the **Scanner** class to create an object to read input from **System.in**, as follows:

```
Scanner input = new Scanner(System.in);
```

The syntax **new Scanner(System.in)** creates an object of the **Scanner** type. The syntax **Scanner input** declares that **input** is a variable whose type is **Scanner**. The whole line **Scanner input = new Scanner(System.in)** creates a **Scanner** object and assigns its reference to the variable **input**. An object may invoke its methods. To invoke a method on an object is to ask the object to perform a task. You can invoke the **nextDouble()** method to read a **double** value as follows:

```
double radius = input.nextDouble();
```

This statement reads a number from the keyboard and assigns the number to **radius**.

Listing 2.2 rewrites Listing 2.1 to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConsoleInput {
  public static void main(String[] args) {
    // Create a Scanner object
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
```

import class

create a Scanner

read a double

```
17         radius + " is " + area);
18     }
19   }
```

```
Enter a number for radius: 2.5 ↵Enter
The area for the circle of radius 2.5 is 19.6349375
```

```
Enter a number for radius: 23 ↵Enter
The area for the circle of radius 23.0 is 1661.90111
```

The **Scanner** class is in the **java.util** package. It is imported in line 1. Line 6 creates a **Scanner** object. Note the **import** statement can be omitted if you replace **Scanner** by **java.util.Scanner** in line 6.

prompt

Line 9 displays a string **"Enter a number for radius: "** to the console. This is known as a *prompt*, because it directs the user to enter an input. Your program should always tell the user what to enter when expecting input from the keyboard.

Recall that the **print** method in line 9 is identical to the **println** method, except that **println** moves to the beginning of the next line after displaying the string, but **print** does not advance to the next line when completed.

Line 6 creates a **Scanner** object. The statement in line 10 reads input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the *Enter* key, the program reads the number and assigns it to **radius**.

More details on objects will be introduced in Chapter 9. *For the time being, simply accept that this is how we obtain input from the console.*

specific import

The **Scanner** class is in the **java.util** package. It is imported in line 1. There are two types of **import** statements: *specific import* and *wildcard import.* The *specific import* specifies a single class in the import statement. For example, the following statement imports **Scanner** from the package **java.util.**

```
import java.util.Scanner;
```

wildcard import

The *wildcard import* imports all the classes in a package by using the asterisk as the wildcard. For example, the following statement imports all the classes from the package **java.util.**

```
import java.util.*;
```

no performance difference

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

Listing 2.3 gives an example of reading multiple inputs from the keyboard. The program reads three numbers and displays their average.

Listing 2.3 ComputeAverage.java

import class

create a Scanner

```
1  import java.util.Scanner; // Scanner is in the java.util package
2
3  public class ComputeAverage {
4    public static void main(String[] args) {
5      // Create a Scanner object
6      Scanner input = new Scanner(System.in);
7
8      // Prompt the user to enter three numbers
9      System.out.print("Enter three numbers: ");
```

```
10      double number1 = input.nextDouble();
11      double number2 = input.nextDouble();
12      double number3 = input.nextDouble();
13
14      // Compute average
15      double average = (number1 + number2 + number3) / 3;
16
17      // Display results
18      System.out.println("The average of " + number1 + " " + number2
19        + " " + number3 + " is " + average);
20    }
21  }
```

read a double

```
Enter three numbers: 1 2 3 ↵Enter
The average of 1.0 2.0 3.0 is 2.0
```

enter input in one line

```
Enter three numbers: 10.5 ↵Enter
11 ↵Enter
11.5 ↵Enter
The average of 10.5 11.0 11.5 is 11.0
```

enter input in multiple lines

The codes for importing the **`Scanner`** class (line 1) and creating a **`Scanner`** object (line 6) are the same as in the preceding example, as well as in all new programs you will write for reading input from the keyboard.

Line 9 prompts the user to enter three numbers. The numbers are read in lines 10–12. You may enter three numbers separated by spaces, then press the *Enter* key, or enter each number followed by a press of the *Enter* key, as shown in the sample runs of this program.

runtime error

If you entered an input other than a numeric value, a runtime error would occur. In Chapter 12, you will learn how to handle the exception so the program can continue to run.

Note

IPO

Most of the programs in the early chapters of this book perform three steps— input, process, and output—called *IPO*. Input is receiving input from the user; process is producing results using the input; and output is displaying the results.

Note

Warning in IDE

If you use an IDE such as Eclipse or NetBeans, you will get a warning to ask you to close the input for preventing a potential resource leak. Ignore the warning for the time being because the input is automatically closed when your program is terminated. In this case, there will be no resource leaking.

Check Point

2.3.1 How do you write a statement to let the user enter a double value from the keyboard? What happens if you entered **`5a`** when executing the following code?

```
double radius = input.nextDouble();
```

2.3.2 Are there any performance differences between the following two **`import`** statements?

```
import java.util.Scanner;
import java.util.*;
```

2.4 Identifiers

Key Point

Identifiers are the names that identify the elements such as classes, methods, and variables in a program.

As you see in Listing 2.3, `ComputeAverage`, `main`, `input`, `number1`, `number2`, `number3`, and so on are the names of things that appear in the program. In programming terminology, such names are called *identifiers*. All identifiers must obey the following rules:

identifiers
identifier naming rules

- An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (`$`).
- An identifier must start with a letter, an underscore (_), or a dollar sign (`$`). It cannot start with a digit.
- An identifier cannot be a reserved word. See Appendix A for a list of reserved words. Reserved words have specific meaning in the Java language. Keywords are reserved words.
- An identifier can be of any length.

For example, `$2`, `ComputeArea`, `area`, `radius`, and `print` are legal identifiers, whereas `2A` and `d+4` are not because they do not follow the rules. The Java compiler detects illegal identifiers and reports syntax errors.

case sensitive

Note

Since Java is case sensitive, `area`, `Area`, and `AREA` are all different identifiers.

descriptive names

Tip

Identifiers are for naming variables, methods, classes, and other items in a program. Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`. We use descriptive names for complete programs in the text. However, we will occasionally use variable names such as `i`, `j`, `k`, `x`, and `y` in the code snippets for brevity. These names also provide a generic tone to the code snippets.

the $ character

Tip

Do not name identifiers with the `$` character. By convention, the `$` character should be used only in mechanically generated source code.

Check Point

2.4.1 Which of the following identifiers are valid? Which are Java keywords?

```
miles, Test, a++, --a, 4#R, $4, #44, apps
class, public, int, x, y, radius
```

2.5 Variables

Key Point

Variables are used to represent values that may be changed in the program.

why called variables?

As you see from the programs in the preceding sections, variables are used to store values to be used later in a program. They are called variables because their values can be changed. In the program in Listing 2.2, `radius` and `area` are variables of the `double` type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, in the following code, `radius` is initially `1.0` (line 2) then changed to `2.0` (line 7), and area is set to `3.14159` (line 3) then reset to `12.56636` (line 8).

```
// Compute the first area
radius = 1.0;                                               radius: 1.0
area = radius * radius * 3.14159;                             area: 3.14159
System.out.println("The area is " + area + " for radius " + radius);

// Compute the second area
radius = 2.0;                                               radius: 2.0
area = radius * radius * 3.14159;                             area: 12.56636
System.out.println("The area is " + area + " for radius " + radius);
```

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
datatype variableName;
```

declare variable

Here are some examples of variable declarations:

```
int count;             // Declare count to be an integer variable
double radius;         // Declare radius to be a double variable
double interestRate;   // Declare interestRate to be a double variable
```

These examples use the data types `int` and `double`. Later you will be introduced to additional data types, such as `byte`, `short`, `long`, `float`, `char`, and `boolean`.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2, ..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```

initialize variables

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```
int count = 1;
```

This is equivalent to the next two statements:

```
int count;
count = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```
int i = 1, j = 2;
```

Tip

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

Every variable has a scope. The *scope of a variable* is the part of the program where the variable can be referenced. The rules that define the scope of a variable will be gradually introduced later in the book. For now, all you need to know is that a variable must be declared and initialized before it can be used.

Check Point

2.5.1 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = k + 2;
    System.out.println(i);
  }
}
```

2.6 Assignment Statements and Assignment Expressions

Key Point

An assignment statement assigns a value to a variable. An assignment statement can also be used as an expression in Java.

assignment statement

assignment operator

After a variable is declared, you can assign a value to it by using an *assignment statement*. In Java, the equal sign (**=**) is used as the *assignment operator*. The syntax for assignment statements is as follows:

```
variable = expression;
```

expression

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. In an assignment statement, the expression on the right-hand side of the assignment operator is evaluated, and then the value is assigned to the variable on the left-hand side of the assignment operator. For example, consider the following code:

```
int y = 1;                 // Assign 1 to variable y
double radius = 1.0;       // Assign 1.0 to variable radius
int x = 5 * (3 / 2);       // Assign the value of the expression to x
x = y + 1;                 // Assign the addition of y and 1 to x
double area = radius * radius * 3.14159;     // Compute area
```

You can use a variable in an expression. A variable can also be used in both sides of the **=** operator. For example,

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

In this assignment statement, the result of **x + 1** is assigned to **x**. If **x** is **1** before the statement is executed, then it becomes **2** after the statement is executed.

To assign a value to a variable, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong:

```
1 = x; // Wrong
```

Note

In mathematics, **x = 2 * x + 1** denotes an equation. However, in Java, **x = 2 * x + 1** is an assignment statement that evaluates the expression **2 * x + 1** and assigns the result to **x**.

assignment expression

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression*. For example, the following statement is correct:

```
System.out.println(x = 1);
```

which is equivalent to

```
x = 1;
System.out.println(x);
```

If a value is assigned to multiple variables, you can use chained assignments like this:

```
i = j = k = 1;
```

which is equivalent to

```
k = 1;
j = k;
i = j;
```

Note

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, **`int x = 1.0`** would be illegal, because the data type of **`x`** is **`int`**. You cannot assign a **`double`** value (**`1.0`**) to an **`int`** variable without using type casting. Type casting will be introduced in Section 2.15.

Check Point

2.6.1 Identify and fix the errors in the following code:

```
public class Test {
  public static void main(String[] args) {
    int i = j = k = 2;
    System.out.println(i + " " + j + " " + k);
  }
}
```

2.7 Named Constants

A named constant is an identifier that represents a permanent value.

Key Point

The value of a variable may change during the execution of a program, but a *named constant,* or simply *constant*, represents permanent data that never changes. A constant is also known as a *final variable* in Java. In our **`ComputeArea`** program, π is a constant. If you use it frequently, you don't want to keep typing **`3.14159`**; instead, you can declare a constant for π. Here is the syntax for declaring a constant:

constant

```
final datatype CONSTANTNAME = value;
```

A constant must be declared and initialized in the same statement. The word **`final`** is a Java keyword for declaring a constant. By convention, all letters in a constant are in uppercase. For example, you can declare π as a constant and rewrite Listing 2.2, as in Listing 2.4.

final keyword

Listing 2.4 ComputeAreaWithConstant.java

```
import java.util.Scanner; // Scanner is in the java.util package

public class ComputeAreaWithConstant {
  public static void main(String[] args) {
    final double PI = 3.14159; // Declare a constant

    // Create a Scanner object
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter a radius
    System.out.print("Enter a number for radius: ");
    double radius = input.nextDouble();

    // Compute area
    double area = radius * radius * PI;

    // Display result
```

```
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

benefits of constants

There are three benefits of using constants: (1) you don't have to repeatedly type the same value if it is used multiple times; (2) if you have to change the constant value (e.g., from `3.14` to `3.14159` for `PI`), you need to change it only in a single location in the source code; and (3) a descriptive name for a constant makes the program easy to read.

2.7.1 What are the benefits of using constants? Declare an `int` constant `SIZE` with value `20`.

2.7.2 Translate the following algorithm into Java code:

Step 1: Declare a `double` variable named `miles` with an initial value `100`.

Step 2: Declare a `double` constant named `KILOMETERS_PER_MILE` with value `1.609`.

Step 3: Declare a `double` variable named `kilometers`, multiply `miles` and `KILOMETERS_PER_MILE`, and assign the result to `kilometers`.

Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

2.8 Naming Conventions

Sticking with the Java naming conventions makes your programs easy to read and avoids errors.

Make sure you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. As mentioned earlier, names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

name variables and methods

- Use lowercase for variables and methods—for example, the variables `radius` and `area`, and the method `print`. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variable `numberOfStudents`. This naming style is known as the *camelCase* because the uppercase characters in the name resemble a camel's humps.

name classes

- Capitalize the first letter of each word in a class name—for example, the class names `ComputeArea` and `System`.

name constants

- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

It is important to follow the naming conventions to make your programs easy to read.

Caution

Do not choose class names that are already used in the Java library. For example, since the `System` class is defined in Java, you should not name your class `System`.

2.8.1 What are the naming conventions for class names, method names, constants, and variables? Which of the following items can be a constant, a method, a variable, or a class according to the Java naming conventions?

`MAX_VALUE`, `Test`, `read`, `readDouble`

2.9 Numeric Data Types and Operations

*Java has six numeric types for integers and floating-point numbers with operators +, –, *, /, and %.*

Key Point

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types and operators.

Table 2.1 lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*
`byte`	-2^7 to $2^7 - 1$ (-128 to 127)	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767)	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: $-3.4028235\text{E}+38$ to $-1.4\text{E}-45$ Positive range: $1.4\text{E}-45$ to $3.4028235\text{E}+38$ 6–9 significant digits	32-bit IEEE 754
`double`	Negative range: $-1.7976931348623157\text{E}+308$ to $-4.9\text{E}-324$ Positive range: $4.9\text{E}-324$ to $1.7976931348623157\text{E}+308$ 15–17 significant digits	64-bit IEEE 754

byte type
short type
int type
long type
float type
double type

Note

IEEE 754 is a standard approved by the Institute of Electrical and Electronics Engineers for representing floating-point numbers on computers. The standard has been widely adopted. Java uses the 32-bit **IEEE 754** for the `float` type and the 64-bit **IEEE 754** for the `double` type. The **IEEE 754** standard also defines special floating-point values, which are listed in Appendix E.

integer types

Java uses four types for integers: `byte`, `short`, `int`, and `long`. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable is within a range of a byte, declare the variable as a `byte`. For simplicity and consistency, we will use `int` for integers most of the time in this book.

floating-point types

Java uses two types for floating-point numbers: `float` and `double`. The `double` type is twice as big as `float`, so the `double` is known as *double precision*, and `float` as *single precision*. Normally, you should use the `double` type, because it is more accurate than the `float` type.

2.9.1 Reading Numbers from the Keyboard

You know how to use the `nextDouble()` method in the `Scanner` class to read a double value from the keyboard. You can also use the methods listed in Table 2.2 to read a number of the `byte`, `short`, `int`, `long`, and `float` type.

TABLE 2.2 Methods for `Scanner` Objects

Method	*Description*
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Here are examples for reading values of various types from the keyboard:

```
1  Scanner input = new Scanner(System.in);
2  System.out.print("Enter a byte value: ");
3  byte byteValue = input.nextByte();
4
5  System.out.print("Enter a short value: ");
6  short shortValue = input.nextShort();
7
8  System.out.print("Enter an int value: ");
9  int intValue = input.nextInt();
10
11 System.out.print("Enter a long value: ");
12 long longValue = input.nextLong();
13
14 System.out.print("Enter a float value: ");
15 float floatValue = input.nextFloat();
```

If you enter a value with an incorrect range or format, a runtime error would occur. For example, if you enter a value `128` for line 3, an error would occur because `128` is out of range for a `byte` type integer.

2.9.2 Numeric Operators

operators +, –, *, /, and %

operands

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`–`), multiplication (`*`), division (`/`), and remainder (`%`), as listed in Table 2.3. The *operands* are the values operated by an operator.

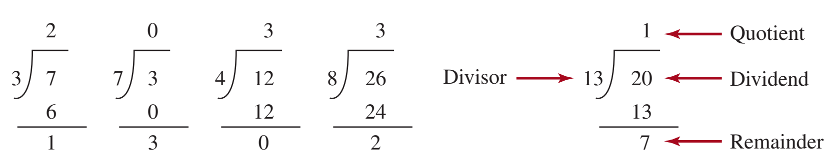
TABLE 2.3 Numeric Operators

Name	*Meaning*	*Example*	*Result*
`+`	Addition	34 + 1	35
`–`	Subtraction	34.0 − 0.1	33.9
`*`	Multiplication	300*30	9000
`/`	Division	1.0 / 2.0	0.5
`%`	Remainder	20 % 3	2

integer division

When both operands of a division are integers, the result of the division is the quotient and the fractional part is truncated. For example, `5 / 2` yields `2`, not `2.5`, and `–5 / 2` yields `–2`, not `–2.5`. To perform a floating-point division, one of the operands must be a floating-point number. For example, `5.0 / 2` yields `2.5`.

The `%` operator, known as *remainder*, yields the remainder after division. The operand on the left is the dividend, and the operand on the right is the divisor. Therefore, `7 % 3` yields `1`, `3 % 7` yields `3`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.



The `%` operator is often used for positive integers, but it can also be used with negative integers and floating-point values. The remainder is negative only if the dividend is negative. For example, `-7 % 3` yields `-1`, `-12 % 4` yields `0`, `-26 % -8` yields `-2`, and `20 % -13` yields `7`.

Remainder is very useful in programming. For example, an even number `% 2` is always `0` and a positive odd number `% 2` is always `1`. Thus, you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What will be the day in 10 days? You can find that the day is Tuesday using the following expression:

Day 6 in a week is Saturday

A week has 7 days

(6 + 10) % 7 is 2

Day 2 in a week is Tuesday
Note: Day 0 is a week is Sunday

After 10 days

The program in Listing 2.5 obtains minutes and remaining seconds from an amount of time in seconds. For example, `500` seconds contains `8` minutes and `20` seconds.

Listing 2.5 DisplayTime.java

```
1  import java.util.Scanner;
2
3  public class DisplayTime {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      // Prompt the user for input
7      System.out.print("Enter an integer for seconds: ");
8      int seconds = input.nextInt();
9
10     int minutes = seconds / 60; // Find minutes in seconds
11     int remainingSeconds = seconds % 60; // Seconds remaining
12     System.out.println(seconds + " seconds is " + minutes +
13       " minutes and " + remainingSeconds + " seconds");
14   }
15 }
```

import Scanner

create a Scanner

read an integer

divide

remainder

```
Enter an integer for seconds: 500 ↵Enter
500 seconds is 8 minutes and 20 seconds
```

line#	seconds	minutes	remainingSeconds
8	500		
10		8	
11			20

The `nextInt()` method (line 8) reads an integer for `seconds`. Line 10 obtains the minutes using `seconds / 60`. Line 11 (`seconds % 60`) obtains the remaining seconds after taking away the minutes.

unary operator

binary operator

The **+** and **–** operators can be both unary and binary. A *unary operator* has only one operand; a *binary operator* has two. For example, the **–** operator in **–5** is a unary operator to negate number **5**, whereas the **–** operator in **4 – 5** is a binary operator for subtracting **5** from **4**.

2.9.3 Exponent Operations

Math.pow(a, b) method

The **Math.pow(a, b)** method can be used to compute a^b. The **pow** method is defined in the **Math** class in the Java API. You invoke the method using the syntax **Math.pow(a, b)** (e.g., **Math.pow(2, 3)**), which returns the result of a^b (2^3). Here, **a** and **b** are parameters for the **pow** method and the numbers **2** and **3** are actual values used to invoke the method. For example,

```
System.out.println(Math.pow(2, 3)); // Displays 8.0
System.out.println(Math.pow(4, 0.5)); // Displays 2.0
System.out.println(Math.pow(2.5, 2)); // Displays 6.25
System.out.println(Math.pow(2.5, -2)); // Displays 0.16
```

Chapter 6 introduces more details on methods. For now, all you need to know is how to invoke the **pow** method to perform the exponent operation.

Check Point

2.9.1 Find the largest and smallest **byte**, **short**, **int**, **long**, **float**, and **double**. Which of these data types requires the least amount of memory?

2.9.2 Show the result of the following remainders:

```
 56 %  6
 78 % -4
-34 %  5
-34 % -5
  5 %  1
  1 %  5
```

2.9.3 If today is Tuesday, what will be the day in 100 days?

2.9.4 What is the result of **25 / 4**? How would you rewrite the expression if you wished the result to be a floating-point number?

2.9.5 Show the result of the following code:

```
System.out.println(2 * (5 / 2 + 5 / 2));
System.out.println(2 * 5 / 2 + 2 * 5 / 2);
System.out.println(2 * (5 / 2));
System.out.println(2 * 5 / 2);
```

2.9.6 Are the following statements correct? If so, show the output.

```
System.out.println("25 / 4 is " + 25 / 4);
System.out.println("25 / 4.0 is " + 25 / 4.0);
System.out.println("3 * 2 / 4 is " + 3 * 2 / 4);
System.out.println("3.0 * 2 / 4 is " + 3.0 * 2 / 4);
```

2.9.7 Write a statement to display the result of $2^{3.5}$.

2.9.8 Suppose **m** and **r** are integers. Write a Java expression for mr^2 to obtain a floating-point result.

Key Point

2.10 Numeric Literals

A literal is a constant value that appears directly in a program.

literal

For example, **34** and **0.305** are literals in the following statements:

```
int numberOfYears = 34;
double weight = 0.305;
```

2.10.1 Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement `byte b = 128`, for example, will cause a compile error, because `128` cannot be stored in a variable of the `byte` type. (Note the range for a byte value is from `-128` to `127`.)

An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (-2147483648) and $2^{31}-1$ (2147483647). To denote an integer literal of the `long` type, append the letter `L` or `l` to it. For example, to write integer `2147483648` in a Java program, you have to write it as `2147483648L` or `2147483648l`, because `2147483648` exceeds the range for the `int` value. `L` is preferred because `l` (lowercase `L`) can easily be confused with 1 (the digit one).

Note

By default, an integer literal is a decimal integer number. To denote a binary integer literal, use a leading `0b` or `0B` (zero B); to denote an octal integer literal, use a leading `0` (zero); and to denote a hexadecimal integer literal, use a leading `0x` or `0X` (zero X). For example,

binary, octal, and hex literals

```
System.out.println(0B1111); // Displays 15
System.out.println(07777); // Displays 4095
System.out.println(0XFFFF); // Displays 65535
```

Hexadecimal numbers, binary numbers, and octal numbers will be introduced in Appendix F.

2.10.2 Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a `double` type value. For example, `5.0` is considered a `double` value, not a `float` value. You can make a number a `float` by appending the letter `f` or `F`, and you can make a number a `double` by appending the letter `d` or `D`. For example, you can use `100.2f` or `100.2F` for a `float` number, and `100.2d` or `100.2D` for a `double` number.

suffix f or F

suffix d or D

Note

The `double` type values are more accurate than the `float` type values. For example,

double vs. float

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
displays 1.0 / 3.0 is 0.3333333333333333
```

16 digits

```
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
displays 1.0F / 3.0F is 0.33333334
```

8 digits

A float value has `6–9` numbers of significant digits, and a double value has `15–17` numbers of significant digits.

Note

To improve readability, Java allows you to use underscores to separate two digits in a number literal. For example, the following literals are correct.

```
long value = 232_45_4519;
double amount = 23.24_4545_4519_3415;
```

However, `45_` or `_45` is incorrect. The underscore must be placed between two digits.

underscores in numbers

2.10.3 Scientific Notation

Floating-point literals can be written in scientific notation in the form of $a \times 10^b$. For example, the scientific notation for 123.456 is 1.23456×10^2 and for 0.0123456 is 1.23456×10^{-2}. A special syntax is used to write scientific notation numbers. For example, 1.23456×10^2 is written as `1.23456E2` or `1.23456E+2` and 1.23456×10^{-2} as `1.23456E-2`. `E` (or `e`) represents an exponent, and can be in either lowercase or uppercase.

why called floating-point?

Note

The `float` and `double` types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation internally. When a number such as `50.534` is converted into scientific notation, such as `5.0534E+1`, its decimal point is moved (i.e., floated) to a new position.

Check Point

2.10.1 How many accurate digits are stored in a `float` or `double` type variable?

2.10.2 Which of the following are correct literals for floating-point numbers?

`12.3`, `12.3e+2`, `23.4e-2`, `-334.4`, `20.5`, `39F`, `40D`

2.10.3 Which of the following are the same as `52.534`?

`5.2534e+1`, `0.52534e+2`, `525.34e-1`, `5.2534e+0`

2.10.4 Which of the following are correct literals?

`5_2534e+1, _2534, 5_2, 5_`

2.11 JShell

Key Point

JShell is a command line tool for quickly evaluating an expression and executing a statement.

JShell is a command line interactive tool introduced in Java 9. JShell enables you to type a single Java statement and get it executed to see the result right away without having to write a complete class. This feature is commonly known as REPL (Read-Evaluate-Print Loop), which evaluates expressions and executes statements as they are entered and shows the result immediately. To use JShell, you need to install JDK 9 or higher. Make sure that you set the correct path on the Windows environment if you use Windows. Open a Command Window and type jshell to launch JShell as shown in Figure 2.2.

FIGURE 2.2 JShell is launched.

You can enter a Java statement from the jshell prompt. For example, enter `int x = 5`, as shown in Figure 2.3.

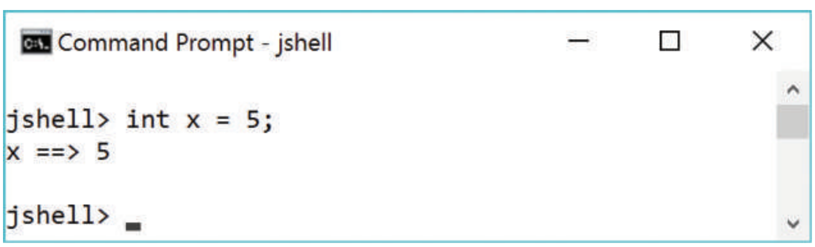


FIGURE 2.3 Enter a Java statement at the jshell command prompt

To print the variable, simply type `x`. Alternatively, you can type `System.out.println(x)`, as shown in Figure 2.4.

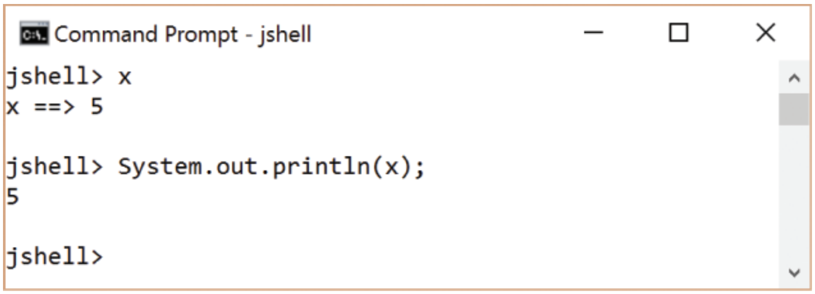


FIGURE 2.4 Print a variable

You can list all the declared variables using the `/vars` command as shown in Figure 2.5.

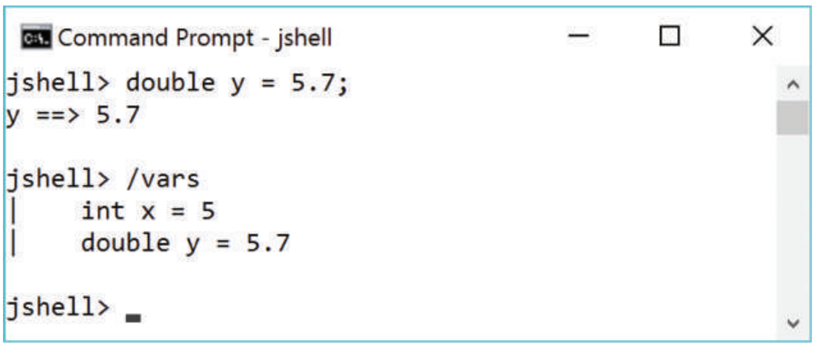
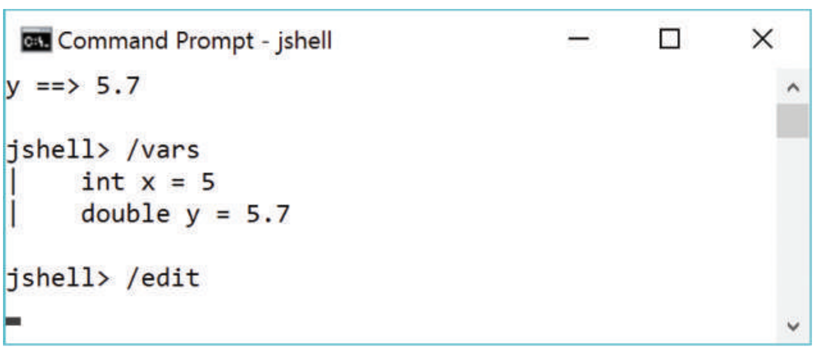
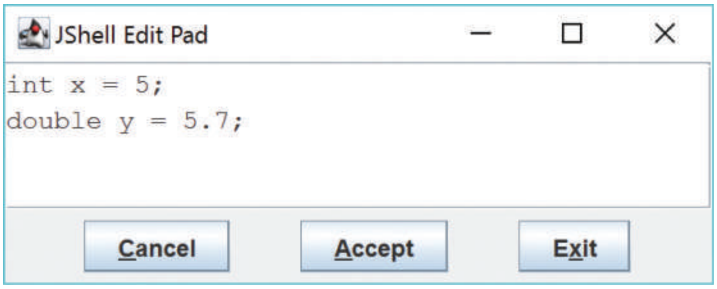


FIGURE 2.5 List all variables

You can use the `/edit` command to edit the code you have entered from the jshell prompt, as shown in Figure 2.6a. This command opens up an edit pane. You can also add/delete the code from the edit pane, as shown in Figure 2.6b. After finishing editing, click the Accept button to make the change in JShell and click the Exit button to exit the edit pane.



(a)



(b)

FIGURE 2.6 The /edit command opens up the edit pane

In JShell, if you don't specify a variable for a value, JShell will automatically create a variable for the value. For example, if you type 6.8 from the jshell prompt, you will see variable $7 is automatically created for 6.8, as shown in Figure 2.7.

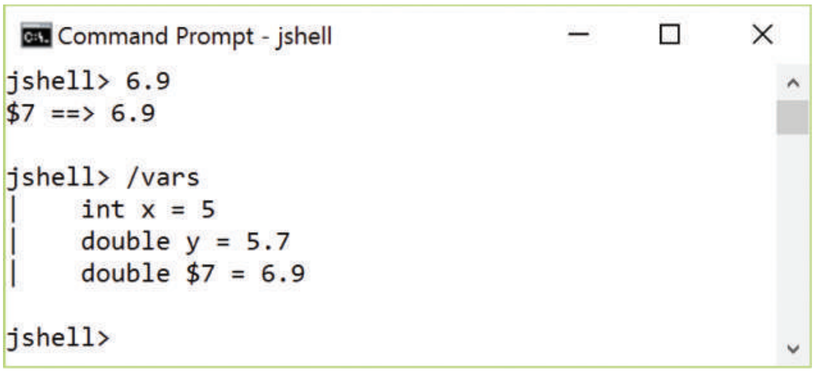


FIGURE 2.7 A variable is automatically created for a value.

To exit JShell, enter /exit.

For more information on JShell, see https://docs.oracle.com/en/java/javase/11/jshell/.

Check Point

2.11.1 What does REPL stand for? How do you launch JShell?

2.12 Evaluating Expressions and Operator Precedence

Key Point

Java expressions are evaluated in the same way as arithmetic expressions.

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithmetic expression

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9 + x}{y}\right)$$

can be translated into a Java expression as follows:

```
(3 + 4 * x) / 5 – 10 * (y - 5) * (a + b + c) / x +
 9 * (4 / x + (9 + x) / y)
```

evaluating an expression

Although Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression is the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation:

operator precedence rule

- Multiplication, division, and remainder operators are applied first. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right.
- Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right.

Here is an example of how an expression is evaluated:

```
3 + 4 * 4 + 5 * (4 + 3) - 1
                                  (1) inside parentheses first
3 + 4 * 4 + 5 * 7 - 1
                                  (2) multiplication
3 + 16 + 5 * 7 - 1
                                  (3) multiplication
3 + 16 + 35 - 1
                                  (4) addition
19 + 35 - 1
                                  (5) addition
54 - 1
                                  (6) subtraction
53
```

Listing 2.6 gives a program that converts a Fahrenheit degree to Celsius using the formula Celsius $= (\frac{5}{9})(\text{Fahrenheit} - 32)$.

LISTING 2.6 FahrenheitToCelsius.java

```java
1   import java.util.Scanner;
2
3   public class FahrenheitToCelsius {
4     public static void main(String[] args) {
5       Scanner input = new Scanner(System.in);
6
7       System.out.print("Enter a degree in Fahrenheit: ");
8       double fahrenheit = input.nextDouble();
9
10      // Convert Fahrenheit to Celsius
11      double celsius = (5.0 / 9) * (fahrenheit - 32);
12      System.out.println("Fahrenheit " + fahrenheit + " is " +
13        celsius + " in Celsius");
14    }
15  }
```

divide

```
Enter a degree in Fahrenheit: 100 ↵Enter
Fahrenheit 100.0 is 37.77777777777778 in Celsius
```

line#	fahrenheit	celsius
8	100	
11		37.77777777777778

integer vs. floating-point division

Be careful when applying division. Division of two integers yields an integer in Java. $\frac{5}{9}$ is coded `5.0 / 9` instead of `5 / 9` in line 11, because `5 / 9` yields `0` in Java.

Check Point

2.12.1 How would you write the following arithmetic expressions in Java?

a. $$\frac{4}{3(r + 34)} - 9(a + bc) + \frac{3 + d(2 + a)}{a + bd}$$

b. $5.5 \times (r + 2.5)^{2.5+t}$

2.13 Case Study: Displaying the Current Time

Key Point

You can invoke `System.currentTimeMillis()` *to return the current time.*

VideoNote

Use operators / and %

The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.

currentTimeMillis

UNIX epoch

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since the time midnight, January 1, 1970 GMT, as shown in Figure 2.8. This time is known as the *UNIX epoch*. The epoch is the point when time starts, and `1970` was the year when the UNIX operating system was formally introduced.

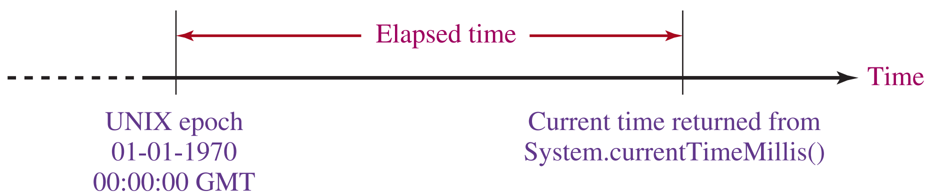


FIGURE 2.8 The `System.currentTimeMillis()` returns the number of milliseconds since the UNIX epoch.

You can use this method to obtain the current time, then compute the current second, minute, and hour as follows:

1. Obtain the total milliseconds since midnight, January 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183068328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183068328` milliseconds / `1000` = `1203183068` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183068` seconds % `60` = `8`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183068` seconds / `60` = `20053051` minutes).
5. Compute the current minute from `totalMinutes % 60` (e.g., `20053051` minutes % `60` = `31`, which is the current minute).
6. Obtain the total hours `totalHours` by dividing `totalMinutes` by `60` (e.g., `20053051` minutes / `60` = `334217` hours).
7. Compute the current hour from `totalHours % 24` (e.g., `334217` hours % `24` = `17`, which is the current hour).

Listing 2.7 gives the complete program.

LISTING 2.7 ShowCurrentTime.java

```
1  public class ShowCurrentTime {
2    public static void main(String[] args) {
3      // Obtain the total milliseconds since midnight, Jan 1, 1970
4      long totalMilliseconds = System.currentTimeMillis();
5
6      // Obtain the total seconds since midnight, Jan 1, 1970
7      long totalSeconds = totalMilliseconds / 1000;
8
9      // Compute the current second in the minute in the hour
10     long currentSecond = totalSeconds % 60;
11
12     // Obtain the total minutes
13     long totalMinutes = totalSeconds / 60;
14
15     // Compute the current minute in the hour
16     long currentMinute = totalMinutes % 60;
17
18     // Obtain the total hours
19     long totalHours = totalMinutes / 60;
20
21     // Compute the current hour
22     long currentHour = totalHours % 24;
23
24     // Display results
25     System.out.println("Current time is " + currentHour + ":"
26       + currentMinute + ":" + currentSecond + " GMT");
27   }
28 }
```

totalMilliseconds
totalSeconds
currentSecond
totalMinutes
currentMinute
totalHours
currentHour
display output

```
Current time is 17:31:8 GMT
```

Line 4 invokes `System.currentTimeMillis()` to obtain the current time in milliseconds as a `long` value. Thus, all the variables are declared as the long type in this program. The seconds, minutes, and hours are extracted from the current time using the `/` and `%` operators (lines 6–22).

variables \ line#	4	7	10	13	16	19	22
totalMilliseconds	1203183068328						
totalSeconds		1203183068					
currentSecond			8				
totalMinutes				20053051			
currentMinute					31		
totalHours						334217	
currentHour							17

In the sample run, a single digit 8 is displayed for the second. The desirable output would be 08. This can be fixed by using a method that formats a single digit with a prefix 0 (see Programming Exercise 6.37).

The hour displayed in this program is in GMT. Programming Exercise 2.8 enables to display the hour in any time zone.

nanoTime

Java also provides the `System.nanoTime()` method that returns the elapse time in nanoseconds. `nanoTime()` is more precise and accurate than `currentTimeMillis()`.

Check Point

2.13.1 How do you obtain the current second, minute, and hour?

2.14 Augmented Assignment Operators

Key Point

The operators **+**, **–**, *****, **/**, *and* **%** *can be combined with the assignment operator to form augmented operators.*

Very often, the current value of a variable is used, modified, then reassigned back to the same variable. For example, the following statement increases the variable `count` by 1:

```
count = count + 1;
```

Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as

```
count += 1;
```

addition assignment operator

The `+=` is called the *addition assignment operator*. Table 2.4 shows other augmented assignment operators.

Table 2.4 Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`–=`	Subtraction assignment	`i –= 8`	`i = i – 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

The augmented assignment operator is performed last after all the other operators in the expression are evaluated. For example,

```
x /= 4 + 5.5 * 1.5;
```

is same as

```
x = x / (4 + 5.5 * 1.5);
```

Caution

There are no spaces in the augmented assignment operators. For example, `+ =` should be `+=`.

Note

Like the assignment operator (`=`), the operators (`+=`, `–=`, `*=`, `/=`, and `%=`) can be used to form an assignment statement as well as an expression. For example, in the following code, `x += 2` is a statement in the first line, and an expression in the second line:

```
x += 2; // Statement
System.out.println(x += 2); // Expression
```

2.14.1 Show the output of the following code:

Check Point

```
double a = 6.5;
a += a + 1;
System.out.println(a);
a = 6;
a /= 2;
System.out.println(a);
```

2.15 Increment and Decrement Operators

The increment operator (++) and decrement operator (−−) are for incrementing and decrementing a variable by 1.

Key Point

increment operator (+ +)
decrement operator (−−)

The `++` and `−−` are two shorthand operators for incrementing and decrementing a variable by `1`. These are handy because that's often how much the value needs to be changed in many programming tasks. For example, the following code increments `i` by `1` and decrements `j` by `1`.

```
int i = 3, j = 3;
i++; // i becomes 4
j−−; // j becomes 2
```

postincrement
postdecrement

`i++` is pronounced as "`i` plus plus" and `i−−` as "`i` minus minus." These operators are known as *postfix increment* (or *postincrement*) and *postfix decrement* (or *postdecrement*), because the operators `++` and `−−` are placed after the variable. These operators can also be placed before the variable. For example,

```
int i = 3, j = 3;
++i; // i becomes 4
−−j; // j becomes 2
```

preincrement
predecrement

`++i` increments `i` by `1` and `−−j` decrements `j` by `1`. These operators are known as *prefix increment* (or *preincrement*) and *prefix decrement* (or predecrement).

As you see, the effect of `i++` and `++i` or `i−−` and `−−i` are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement. Table 2.5 describes their differences and gives examples.

TABLE 2.5 Increment and Decrement Operators

Operator	*Name*	*Description*	*Example* (assume i = 1)
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` // j is 2, i is 2
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` // j is 1, i is 2
`−−var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = −−i;` // j is 0, i is 0
`var−−`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i−−;` // j is 1, i is 0

Here are additional examples to illustrate the differences between the prefix form of `++` (or `−−`) and the postfix form of `++` (or `−−`). Consider the following code:

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Output is

i is 11, newNum is 100

In this case, `i` is incremented by `1`, then the *old* value of `i` is used in the multiplication. Thus, `newNum` becomes `100`. If `i++` is replaced by `++i`, then it becomes as follows:

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

Output is

i is 11, newNum is 110

`i` is incremented by `1`, and the new value of `i` is used in the multiplication. Thus, `newNum` becomes `110`.

Here is another example:

```
double x = 1.0;
double y = 5.0;
double z = x–– + (++y);
```

After all three lines are executed, `y` becomes `6.0`, `z` becomes `7.0`, and `x` becomes `0.0`.

Operands are evaluated from left to right in Java. The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated. This rule takes precedence over any other rules that govern expressions. Here is an example:

```
int i = 1;
int k = ++i + i * 3;
```

`++i` is evaluated and returns `2`. When evaluating `i * 3`, `i` is now `2`. Therefore, `k` becomes `8`.

Tip

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i * 3`.

Check Point

2.15.1 Which of these statements are true?

a. Any expression can be used as a statement.

b. The expression `x++` can be used as a statement.

c. The statement `x = x + 5` is also an expression.

d. The statement `x = y = x = 0` is illegal.

2.15.2 Show the output of the following code:

```
int a = 6;
int b = a++;
System.out.println(a);
System.out.println(b);
a = 6;
b = ++a;
System.out.println(a);
System.out.println(b);
```

2.16 Numeric Type Conversions

Key Point

Floating-point numbers can be converted into integers using explicit casting.

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. Therefore, `3 * 4.5` is the same as `3.0 * 4.5`.

You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a **long** value to a **float** variable. You cannot, however, assign a value to a variable of a type with a smaller range unless you use *type casting*. *Casting* is an operation that converts a value of one data type into a value of another data type. Casting a type with a small range to a type with a larger range is known as *widening a type*. Casting a type with a large range to a type with a smaller range is known as *narrowing a type*. Java will automatically widen a type, but you must narrow a type explicitly.

casting

widening a type

narrowing a type

The syntax for casting a type is to specify the target type in parentheses, followed by the variable's name or the value to be cast. For example, the following statement

```
System.out.println((int)1.7);
```

displays **1**. When a **double** value is cast into an **int** value, the fractional part is truncated.

The following statement

```
System.out.println((double)1 / 2);
```

displays **0.5**, because **1** is cast to **1.0** first, then **1.0** is divided by **2**. However, the statement

```
System.out.println(1 / 2);
```

displays **0**, because **1** and **2** are both integers and the resulting value should also be an integer.

Caution

Casting is necessary if you are assigning a value to a variable of a smaller type range, such as assigning a **double** value to an **int** variable. A compile error will occur if casting is not used in situations of this kind. However, be careful when using casting, as loss of information might lead to inaccurate results.

possible loss of precision

Note

Casting does not change the variable being cast. For example, **d** is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d; // i becomes 4, but d is still 4.5
```

Note

In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct:

casting in an augmented expression

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement
```

sum += 4.5 is equivalent to **sum = (int)(sum + 4.5)**.

Note

To assign a variable of the **int** type to a variable of the **short** or **byte** type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the **short** or **byte** type (see Section 2.10, Numeric Literals).

The program in Listing 2.8 displays the sales tax with two digits after the decimal point.

LISTING 2.8 SalesTax.java

casting

```
1  import java.util.Scanner;
2
3  public class SalesTax {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      System.out.print("Enter purchase amount: ");
8      double purchaseAmount = input.nextDouble();
9
10     double tax = purchaseAmount * 0.06;
11     System.out.println("Sales tax is $" + (int)(tax * 100) / 100.0);
12   }
13 }
```

```
Enter purchase amount: 197.55 ↵Enter
Sales tax is $11.85
```

line#	purchaseAmount	tax	Output
8	197.55		
10		11.853	
11			11.85

formatting numbers

Using the input in the sample run, the variable `purchaseAmount` is `197.55` (line 8). The sales tax is `6%` of the purchase, so the `tax` is evaluated as `11.853` (line 10). Note

`tax * 100` is `1185.3`
`(int)(tax * 100)` is `1185`
`(int)(tax * 100) / 100.0` is `11.85`

Thus, the statement in line 11 displays the tax `11.85` with two digits after the decimal point. Note the expression `(int)(tax * 100) / 100.0` rounds down `tax` to two decimal places. If `tax` is `3.456`, `(int)(tax * 100) / 100.0` would be `3.45`. Can it be rounded up to two decimal places? Note any double value `x` can be rounded up to an integer using `(int)(x + 0.5)`. Thus, `tax` can be rounded up to two decimal places using `(int)(tax * 100 + 0.5) / 100.0`.

Check Point

2.16.1 Can different types of numeric values be used together in a computation?

2.16.2 What does an explicit casting from a `double` to an `int` do with the fractional part of the `double` value? Does casting change the variable being cast?

2.16.3 Show the following output:

```
float f = 12.5F;
int i = (int)f;
System.out.println("f is " + f);
System.out.println("i is " + i);
```

2.16.4 If you change `(int)(tax * 100) / 100.0` to `(int)(tax * 100) / 100` in line 11 in Listing 2.8, what will be the output for the input purchase amount of `197.556`?

2.16.5 Show the output of the following code:

```
double amount = 5;
System.out.println(amount / 2);
System.out.println(5 / 2);
```

2.16.6 Write an expression that rounds up a `double` value in variable `d` to an integer.

2.17 Software Development Process

The software development life cycle is a multistage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.

Key Point

Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: requirements specification, analysis, design, implementation, testing, deployment, and maintenance, as shown in Figure 2.9.

VideoNote
Software development process

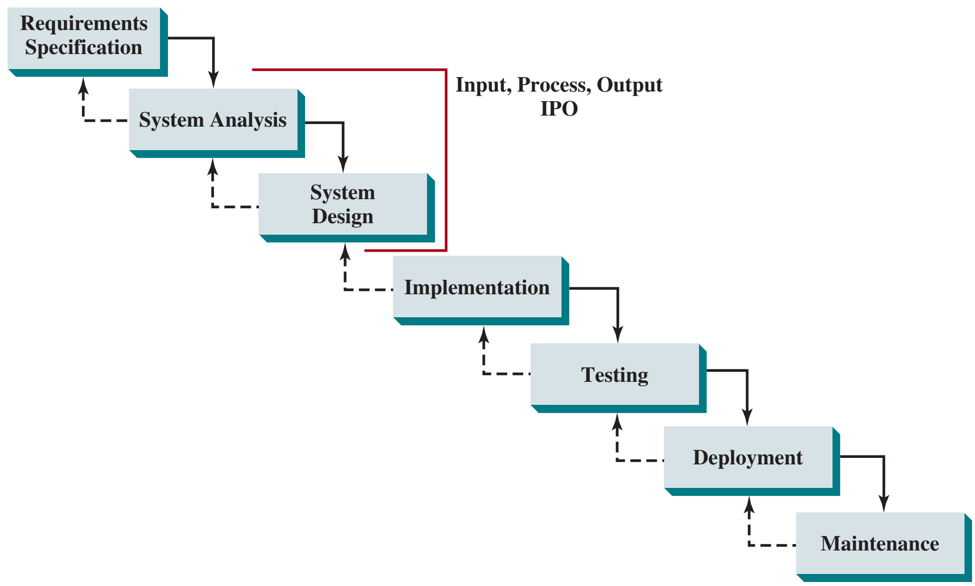


FIGURE 2.9 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

Requirements specification is a formal process that seeks to understand the problem the software will address, and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not always well defined. Developers need to work closely with their customers (the individuals or organizations that will use the software) and study the problem carefully to identify what the software needs to do. requirements specification

System analysis seeks to analyze the data flow and to identify the system's input and output. When you perform analysis, it helps to identify what the output is first, then figure out what input data you need in order to produce the output. system analysis

System design is to design a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output (IPO). system design IOP

Implementation involves translating the system design into programs. Separate programs are written for each component then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, self-testing, and debugging (that is, finding errors, called *bugs,* in the code). implementation

Testing ensures the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing. testing deployment

Deployment makes the software available for use. Depending on the type of software, it may be installed on each user's machine, or installed on a server accessible on the Internet.

maintenance

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

VideoNote

Compute loan payments

To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

$$monthlyPayment = \frac{loanAmount \times monthlyInterestRate}{1 - \dfrac{1}{(1 + monthlyInterestRate)^{numberOfYears \times 12}}}$$

$$totalPayment = monthlyPayment \times numberOfYears \times 12$$

Therefore, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.

> **Note**
> The requirements specification says the user must enter the annual interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible you may discover that input is not sufficient or some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.

> **Note**
> In the real world, you will work with customers from all walks of life. You may develop software for chemists, physicists, engineers, economists, and psychologists, and of course you will not have (or need) complete knowledge of all these fields. Therefore, you don't have to know how formulas are derived, but given the monthly interest rate, the number of years, and the loan amount, you can compute the monthly payment in this program. You will, however, need to communicate with customers and understand how a mathematical model works for the system.

Stage 3: System Design

During system design, you identify the steps in the program.

Step 3.1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount.

(The interest rate is commonly expressed as a percentage of the principal for a period of one year. This is known as the *annual interest rate*.)

Step 3.2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by **100**. To obtain the monthly interest rate from the annual interest rate, divide it by **12**, since a year has 12 months. Thus, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by **1200**. For example, if the annual interest rate is 4.5%, then the monthly interest rate is $4.5/1200 = 0.00375$.

Step 3.3. Compute the monthly payment using the preceding formula.

Step 3.4. Compute the total payment, which is the monthly payment multiplied by **12** and multiplied by the number of years.

Step 3.5. Display the monthly payment and total payment.

Stage 4: Implementation

Implementation is also known as *coding* (writing the code). In the formula, you have to compute $(1 + \mathtt{monthlyInterestRate})^{\mathtt{numberOfYears} \times 12}$, which can be obtained using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

Math.pow(a, b) method

Listing 2.9 gives the complete program.

LISTING 2.9 ComputeLoan.java

```java
import java.util.Scanner;

public class ComputeLoan {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Enter annual interest rate in percentage, e.g., 7.25
    System.out.print("Enter annual interest rate, e.g., 7.25: ");
    double annualInterestRate = input.nextDouble();

    // Obtain monthly interest rate
    double monthlyInterestRate = annualInterestRate / 1200;

    // Enter number of years
    System.out.print(
      "Enter number of years as an integer, e.g., 5: ");
    int numberOfYears = input.nextInt();

    // Enter loan amount
    System.out.print("Enter loan amount, e.g., 120000.95: ");
    double loanAmount = input.nextDouble();

    // Calculate payment
    double monthlyPayment = loanAmount * monthlyInterestRate / (1
      - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
    double totalPayment = monthlyPayment * numberOfYears * 12;

    // Display results
    System.out.println("The monthly payment is $" +
      (int)(monthlyPayment * 100) / 100.0);
    System.out.println("The total payment is $" +
      (int)(totalPayment * 100) / 100.0);
  }
}
```

import class
create a Scanner
enter interest rate
enter years
enter loan amount
monthlyPayment
totalPayment
casting
casting

```
Enter annual interest rate, for example, 7.25: 5.75 ↵Enter
Enter number of years as an integer, for example, 5: 15 ↵Enter
Enter loan amount, for example, 120000.95: 250000 ↵Enter
The monthly payment is $2076.02
The total payment is $373684.53
```

variables \ line#	10	13	18	22	25	27
annualInterestRate	5.75					
monthlyInterestRate		0.0047916666666				
numberOfYears			15			
loanAmount				250000		
monthlyPayment					2076.0252175	
totalPayment						373684.539

Line 10 reads the annual interest rate, which is converted into the monthly interest rate in line 13.

Choose the most appropriate data type for the variable. For example, `numberOfYears` is best declared as an `int` (line 18), although it could be declared as a `long`, `float`, or `double`. Note `byte` might be the most appropriate for `numberOfYears`. For simplicity, however, the examples in this booktext will use `int` for integer and `double` for floating-point values.

The formula for computing the monthly payment is translated into Java code in lines 25–27.

Casting is used in lines 31 and 33 to obtain a new `monthlyPayment` and `totalPayment` with two digits after the decimal points.

`java.lang` package

The program uses the `Scanner` class, imported in line 1. The program also uses the `Math` class, and you might be wondering why that class isn't imported into the program. The `Math` class is in the `java.lang` package, and all classes in the `java.lang` package are implicitly imported. Therefore, you don't need to explicitly import the `Math` class.

Stage 5: Testing

After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases, as you will see in later chapters. For these types of problems, you need to design test data that cover all cases.

incremental coding and testing

Tip

The system design phase in this example identified several steps. It is a good approach to code and test these steps incrementally by adding them one at a time. This approach, called incremental coding and testing, makes it much easier to pinpoint problems and debug the program.

Check Point

2.17.1 How would you write the following arithmetic expression?

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

2.18 Case Study: Counting Monetary Units

Key Point

This section presents a program that breaks a large amount of money into smaller units.

Suppose you want to develop a program that changes a given amount of money into smaller monetary units. The program lets the user enter an amount as a `double` value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in the maximum number of dollars, quarters, dimes, nickels, and pennies, in this order, to result in the minimum number of coins.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as `11.56`.
2. Convert the amount (e.g., `11.56`) into cents (`1156`).
3. Divide the cents by `100` to find the number of dollars. Obtain the remaining cents using the cents remainder `100`.
4. Divide the remaining cents by `25` to find the number of quarters. Obtain the remaining cents using the remaining cents remainder `25`.
5. Divide the remaining cents by `10` to find the number of dimes. Obtain the remaining cents using the remaining cents remainder `10`.
6. Divide the remaining cents by `5` to find the number of nickels. Obtain the remaining cents using the remaining cents remainder `5`.
7. The remaining cents are the pennies.
8. Display the result.

The complete program is given in Listing 2.10.

LISTING 2.10 ComputeChange.java

```java
import java.util.Scanner;

public class ComputeChange {
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);

    // Receive the amount
    System.out.print(
      "Enter an amount in double, for example 11.56: ");
    double amount = input.nextDouble();

    int remainingAmount = (int)(amount * 100);

    // Find the number of one dollars
    int numberOfOneDollars = remainingAmount / 100;
    remainingAmount = remainingAmount % 100;

    // Find the number of quarters in the remaining amount
    int numberOfQuarters = remainingAmount / 25;
    remainingAmount = remainingAmount % 25;

    // Find the number of dimes in the remaining amount
    int numberOfDimes = remainingAmount / 10;
    remainingAmount = remainingAmount % 10;

    // Find the number of nickels in the remaining amount
    int numberOfNickels = remainingAmount / 5;
    remainingAmount = remainingAmount % 5;

    // Find the number of pennies in the remaining amount
    int numberOfPennies = remainingAmount;

```

import class
enter input
dollars
quarters
dimes
nickels
pennies

output

```
34     // Display results
35     System.out.println("Your amount " + amount + " consists of");
36     System.out.println(" " + numberOfOneDollars + " dollars");
37     System.out.println(" " + numberOfQuarters + " quarters ");
38     System.out.println(" " + numberOfDimes + " dimes");
39     System.out.println(" " + numberOfNickels + " nickels");
40     System.out.println(" " + numberOfPennies + " pennies");
41   }
42 }
```

```
Enter an amount in double, for example, 11.56: 11.56 ↵Enter
Your amount 11.56 consists of
      11 dollars
      2 quarters
      0 dimes
      1 nickels
      1 pennies
```

variables \ line#	11	13	16	17	20	21	24	25	28	29	32
amount	11.56										
remainingAmount		1156		56		6		6		1	
numberOfOneDollars			11								
numberOfQuarters					2						
numberOfDimes							0				
numberOfNickels									1		
numberOfPennies											1

The variable **amount** stores the amount entered from the console (line 11). This variable is not changed, because the amount has to be used at the end of the program to display the results. The program introduces the variable **remainingAmount** (line 13) to store the changing remaining amount.

The variable **amount** is a **double** decimal representing dollars and cents. It is converted to an **int** variable **remainingAmount**, which represents all the cents. For instance, if **amount** is **11.56**, then the initial **remainingAmount** is **1156**. The division operator yields the integer part of the division, so **1156 / 100** is **11**. The remainder operator obtains the remainder of the division, so **1156 % 100** is **56**.

The program extracts the maximum number of singles from the remaining amount and obtains a new remaining amount in the variable **remainingAmount** (lines 16–17). It then extracts the maximum number of quarters from **remainingAmount** and obtains a new **remainingAmount** (lines 20–21). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

loss of precision

One serious problem with this example is the possible loss of precision when casting a **double** amount to an **int remainingAmount**. This could lead to an inaccurate result. If you try to enter the amount **10.03**, **10.03 * 100** becomes **1002.9999999999999**. You will find that the program displays **10** dollars and **2** pennies. To fix the problem, enter the amount as an integer value representing cents (see Programming Exercise 2.22).

Check Point

2.18.1 Show the output of Listing 2.10 with the input value **1.99**. Why does the program produce an incorrect result for the input 10.03?

2.19 Common Errors and Pitfalls

Common elementary programming errors often involve undeclared variables, uninitialized variables, integer overflow, unintended integer division, and round-off errors.

Key Point

Common Error 1: Undeclared/Uninitialized Variables and Unused Variables

A variable must be declared with a type and assigned a value before using it. A common error is not declaring a variable or initializing a variable. Consider the following code:

```
double interestRate = 0.05;
double interest = interestrate * 45;
```

This code is wrong, because `interestRate` is assigned a value `0.05`; but `interestrate` has not been declared and initialized. Java is case sensitive, so it considers `interestRate` and `interestrate` to be two different variables.

If a variable is declared, but not used in the program, it might be a potential programming error. Therefore, you should remove the unused variable from your program. For example, in the following code, `taxRate` is never used. It should be removed from the code.

```
double interestRate = 0.05;
double taxRate = 0.05;
double interest = interestRate * 45;
System.out.println("Interest is " + interest);
```

If you use an IDE such as Eclipse and NetBeans, you will receive a warning on unused variables.

Common Error 2: Integer Overflow

Numbers are stored with a limited numbers of digits. When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes overflow, because the largest value that can be stored in a variable of the `int` type is `2147483647`. `2147483648` will be too large for an `int` value: what is overflow?

```
int value = 2147483647 + 1;
// value will actually be -2147483648
```

Likewise, executing the following statement also causes overflow, because the smallest value that can be stored in a variable of the `int` type is `-2147483648`. `-2147483649` is too large in size to be stored in an `int` variable.

```
int value = -2147483648 - 1;
// value will actually be 2147483647
```

Java does not report warnings or errors on overflow, so be careful when working with integers close to the maximum or minimum range of a given type.

When a floating-point number is too small (i.e., too close to zero) to be stored, it causes *underflow*. Java approximates it to zero, so normally you don't need to be concerned about underflow. what is underflow?

Common Error 3: Round-off Errors

A *round-off error*, also called a *rounding error*, is the difference between the calculated approximation of a number and its exact mathematical value. For example, 1/3 is approximately 0.333 if you keep three decimal places, and is 0.3333333 if you keep seven decimal places. Since the number of digits that can be stored in a variable is limited, round-off errors are inevitable. Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example, floating-point approximation

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

displays `0.5000000000000001`, not `0.5`, and

```
System.out.println(1.0 - 0.9);
```

displays `0.09999999999999998`, not `0.1`. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

Common Error 4: Unintended Integer Division

Java uses the same divide operator, namely `/`, to perform both integer and floating-point division. When two operands are integers, the `/` operator performs an integer division. The result of the operation is an integer. The fractional part is truncated. To force two integers to perform a floating-point division, make one of the integers into a floating-point number. For example, the code in (a) displays that `average` as `1` and the code in (b) displays that `average` as `1.5`.

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

(a)

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2.0;
System.out.println(average);
```

(b)

Common Pitfall 1: Redundant Input Objects

New programmers often write the code to create multiple input objects for each input. For example, the following code in (a) reads an integer and a double value:

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();

Scanner input1 = new Scanner(System.in);
System.out.print("Enter a double value: ");
double v2 = input1.nextDouble();
```

BAD CODE

The code is not good. It creates two input objects unnecessarily and may lead to some subtle errors. You should rewrite the code in (b):

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int v1 = input.nextInt();
System.out.print("Enter a double value: ");
double v2 = input.nextDouble();
```

GOOD CODE

Check Point

2.19.1 Can you declare a variable as `int` and later redeclare it as `double`?

2.19.2 What is an integer overflow? Can floating-point operations cause overflow?

2.19.3 Will overflow cause a runtime error?

2.19.4 What is a round-off error? Can integer operations cause round-off errors? Can floating-point operations cause round-off errors?

Key Terms

algorithm, 34
assignment operator (=), 42
assignment statement, 42
`byte` *type*, 45
casting, 59
constant, 43
data type, 35
declare variables, 35

decrement operator (−−), 57
`double` *type*, 45
expression, 42
`final` keyword, 43
`float` *type*, 45
floating-point number, 35
identifier, 40
increment operator (++), 57
incremental coding and testing, 64
`int` *type*, 45
IPO, 39
literal, 48
`long` *type*, 45
narrowing a type, 59
operand, 46
operator, 46
overflow, 67
postdecrement, 57
postincrement, 57
predecrement, 57
preincrement, 57
primitive data type, 35
pseudocode, 34
requirements specification, 61
scope of a variable, 41
`short` *type*, 45
specific import, 38
system analysis, 61
system design, 61
underflow, 67
UNIX epoch, 54
variable, 35
widening a type, 59
wildcard import, 38

Chapter Summary

1. *Identifiers* are names for naming elements such as variables, constants, methods, classes, and packages in a program.

2. An identifier is a sequence of characters that consists of letters, digits, underscores (`_`), and dollar signs (`$`). An identifier must start with a letter or an underscore. It cannot start with a digit. An identifier cannot be a reserved word. An identifier can be of any length.

3. *Variables* are used to store data in a program. To declare a variable is to tell the compiler what type of data a variable can hold.

4. There are two types of `import` statements: *specific import* and *wildcard import*. The specific import specifies a single class in the import statement. The wildcard import imports all the classes in a package.

5. In Java, the equal sign (`=`) is used as the *assignment operator.*

6. A variable declared in a method must be assigned a value before it can be used.

7. A *named constant* (or simply a *constant*) represents permanent data that never changes.

8. A named constant is declared by using the keyword `final`.

9. Java provides four integer types (`byte`, `short`, `int`, and `long`) that represent integers of four different sizes.

10. Java provides two *floating-point types* (`float` and `double`) that represent floating-point numbers of two different precisions.

11. Java provides *operators* that perform numeric operations: `+` (addition), `–` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).

12. Integer arithmetic (`/`) yields an integer result.

13. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.

14. Java provides the augmented assignment operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).

15. The *increment operator* (`++`) and the *decrement operator* (`--`) increment or decrement a variable by `1`.

16. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.

17. You can explicitly convert a value from one type to another using the `(type)value` notation.

18. *Casting* a variable of a type with a small range to a type with a larger range is known as *widening a type*.

19. Casting a variable of a type with a large range to a type with a smaller range is known as *narrowing a type.*

20. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

21. In computer science, midnight of January 1, 1970, is known as the *UNIX epoch.*

Quiz

Answer the quiz for this chapter online at the Companion Website.

MyProgrammingLab™

Programming Exercises

learn from examples

Debugging Tip
The compiler usually gives a reason for a syntax error. If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text.

document analysis and design

Pedagogical Note
Instructors may ask you to document your analysis and design for selected exercises. Use your own words to analyze the problem, including the input, output, and what needs to be computed, and describe how to solve the problem in pseudocode.

even-numbered programming exercises

Pedagogical Note
The solution to most even-numbered programming exercises are provided to students. These exercises serve as additional examples for a variety of programs. To maximize the benefits of these solutions, students should first attempt to complete the even-numbered exercises and then compare their solutions with the solutions provided in the book. Since the book provides a large number of programming exercises, it is sufficient if you can complete all even-numbered programming exercises.

Sections 2.2–2.13

2.1 (*Convert Celsius to Fahrenheit*) Write a program that reads a Celsius degree in a `double` value from the console, then converts it to Fahrenheit, and displays the result. The formula for the conversion is as follows:

```
fahrenheit = (9 / 5) * celsius + 32
```

Hint: In Java, `9 / 5` is `1`, but `9.0 / 5` is `1.8`.

Here is a sample run:

```
Enter a degree in Celsius: 43.5 ↵Enter
43.5 Celsius is 110.3 Fahrenheit
```

2.2 (*Compute the volume of a cylinder*) Write a program that reads in the radius and length of a cylinder and computes the area and volume using the following formulas:

```
area = radius * radius * π
volume = area * length
```

Here is a sample run:

```
Enter the radius and length of a cylinder: 5.5 12 ↵Enter
The area is 95.0331
The volume is 1140.4
```

2.3 (*Convert feet into meters*) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is **0.305** meter. Here is a sample run:

```
Enter a value for feet: 16.5 ↵Enter
16.5 feet is 5.0325 meters
```

2.4 (*Convert pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is **0.454** kilogram. Here is a sample run:

```
Enter a number in pounds: 55.5 ↵Enter
55.5 pounds is 25.197 kilograms
```

***2.5** (*Financial application: calculate tips*) Write a program that reads the subtotal and the gratuity rate, then computes the gratuity and total. For example, if the user enters **10** for subtotal and **15%** for gratuity rate, the program displays **$1.5** as gratuity and **$11.5** as total. Here is a sample run:

```
Enter the subtotal and a gratuity rate: 10 15 ↵Enter
The gratuity is $1.5 and total is $11.5
```

****2.6** (*Sum the digits in an integer*) Write a program that reads an integer between **0** and **1000** and adds all the digits in the integer. For example, if an integer is **932**, the sum of all its digits is **14**.

Hint: Use the **%** operator to extract digits, and use the **/** operator to remove the extracted digit. For instance, **932 % 10 = 2** and **932 / 10 = 93**.

Here is a sample run:

```
Enter a number between 0 and 1000: 999 ↵Enter
The sum of the digits is 27
```

***2.7** (*Find the number of years*) Write a program that prompts the user to enter the minutes (e.g., 1 billion), and displays the maximum number of years and remaining days for the minutes. For simplicity, assume that a year has `365` days. Here is a sample run:

```
Enter the number of minutes: 1000000000
1000000000 minutes is approximately 1902 years and 214 days
```

***2.8** (*Current time*) Listing 2.7, ShowCurrentTime.java, gives a program that displays the current time in GMT. Revise the program so it prompts the user to enter the time zone offset to GMT and displays the time in the specified time zone. Here is a sample run:

```
Enter the time zone offset to GMT: -5
The current time is 4:50:34
```

2.9 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as given by the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity v_0 in meters/second, the ending velocity v_1 in meters/second, and the time span t in seconds, then displays the average acceleration. Here is a sample run:

```
Enter v0, v1, and t: 5.5 50.9 4.5
The average acceleration is 10.0889
```

2.10 (*Science: calculating energy*) Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

```
Q = M * (finalTemperature – initialTemperature) * 4184
```

where `M` is the weight of water in kilograms, initial and final temperatures are in degrees Celsius, and energy `Q` is measured in joules. Here is a sample run:

```
Enter the amount of water in kilograms: 55.5
Enter the initial temperature: 3.5
Enter the final temperature: 10.5
The energy needed is 1625484.0
```

2.11 (*Population projection*) Rewrite Programming Exercise 1.11 to prompt the user to enter the number of years and display the population after the number of years. Use the hint in Programming Exercise 1.11 for this program. Here is a sample run of the program:

```
Enter the number of years: 5
The population in 5 years is 325932969
```

2.12 (*Physics: finding runway length*) Given an airplane's acceleration a and take-off speed v, you can compute the minimum runway length needed for an airplane to take off using the following formula:

$$\text{length} = \frac{v^2}{2a}$$

Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), then, displays the minimum runway length.

```
Enter speed and acceleration: 60 3.5
The minimum runway length for this airplane is 514.286
```

****2.13** (*Financial application: compound value*) Suppose you save **`$100`** *each* month into a savings account with an annual interest rate of 5%. Thus, the monthly interest rate is $0.05/12 = 0.00417$. After the first month, the value in the account becomes

```
100 * (1 + 0.00417) = 100.417
```

After the second month, the value in the account becomes

```
(100 + 100.417) * (1 + 0.00417) = 201.252
```

After the third month, the value in the account becomes

```
(100 + 201.252) * (1 + 0.00417) = 302.507
```

and so on.

Write a program that prompts the user to enter a monthly saving amount and displays the account value after the sixth month. (In Programming Exercise 5.30, you will use a loop to simplify the code and display the account value for any month.)

```
Enter the monthly saving amount: 100
After the sixth month, the account value is $608.81
```

***2.14** (*Health application: computing BMI*) Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing, by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and displays the BMI. Note one pound is **`0.45359237`** kilograms and one inch is **`0.0254`** meters. Here is a sample run:

VideoNote
Compute BMI

```
Enter weight in pounds: 95.5
Enter height in inches: 50
BMI is 26.8573
```

2.15 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points **`(x1, y1)`** and **`(x2, y2)`** and displays their distance. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note you can use **`Math.pow(a, 0.5)`** to compute $\sqrt{a}$. Here is a sample run:

```
Enter x1 and y1: 1.5 -3.4
Enter x2 and y2: 4 5
The distance between the two points is 8.764131445842194
```

2.16 (*Geometry: area of a hexagon*) Write a program that prompts the user to enter the side of a hexagon and displays its area. The formula for computing the area of a hexagon is

$$\text{Area} = \frac{3\sqrt{3}}{2} s^2,$$

where s is the length of a side. Here is a sample run:

```
Enter the length of the side: 5.5 ↵Enter
The area of the hexagon is 78.5918
```

***2.17** (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_a v^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit, v is the speed measured in miles per hour, and t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or temperatures below −58°F or above 41°F.

Write a program that prompts the user to enter a temperature between −58°F and 41°F and a wind speed greater than or equal to `2` then displays the wind-chill temperature. Use `Math.pow(a, b)` to compute $v^{0.16}$. Here is a sample run:

```
Enter the temperature in Fahrenheit between -58°F and 41°F:
5.3 ↵Enter
Enter the wind speed (>= 2) in miles per hour: 6 ↵Enter
The wind chill index is -5.56707
```

2.18 (*Print a table*) Write a program that displays the following table. Cast floating-point numbers into integers.

```
a     b     pow(a, b)
1     2     1
2     3     8
3     4     81
4     5     1024
5     6     15625
```

***2.19** (*Geometry: area of a triangle*) Write a program that prompts the user to enter three points, `(x1, y1)`, `(x2, y2)`, and `(x3, y3)`, of a triangle then displays its area. The formula for computing the area of a triangle is

$$s = (\text{side1} + \text{side2} + \text{side3})/2;$$

$$\text{area} = \sqrt{s(s - \text{side1})(s - \text{side2})(s - \text{side3})}$$

Here is a sample run:

```
Enter the coordinates of three points separated by spaces
like x1 y1 x2 y2 x3 y3: 1.5 -3.4 4.6 5 9.5 -3.4 ↵Enter
The area of the triangle is 33.6
```

Sections 2.13–2.18

***2.20** (*Financial application: calculate interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\texttt{annualInterestRate}/1200)$$

Write a program that reads the balance and the annual percentage interest rate and displays the interest for the next month. Here is a sample run:

```
Enter balance and interest rate (e.g., 3 for 3%): 1000 3.5 ↵Enter
The interest is 2.91667
```

***2.21** (*Financial application: calculate future investment value*) Write a program that reads in investment amount, annual interest rate, and number of years and displays the future investment value using the following formula:

$$\texttt{futureInvestmentValue} = \texttt{investmentAmount} \times (1 + \texttt{monthlyInterestRate})^{\texttt{numberOfYears}*12}$$

For example, if you enter amount **`1000`**, annual interest rate **`3.25%`**, and number of years **`1`**, the future investment value is **`1032.98`**.

Here is a sample run:

```
Enter investment amount: 1000.56 ↵Enter
Enter annual interest rate in percentage: 4.25 ↵Enter
Enter number of years: 1 ↵Enter
Future value is $1043.92
```

***2.22** (*Financial application: monetary units*) Rewrite Listing 2.10, ComputeChange.java, to fix the possible loss of accuracy when converting a **`double`** value to an **`int`** value. Enter the input as an integer whose last two digits represent the cents. For example, the input **`1156`** represents **`11`** dollars and **`56`** cents.

***2.23** (*Cost of driving*) Write a program that prompts the user to enter the distance to drive, the fuel efficiency of the car in miles per gallon, and the price per gallon then displays the cost of the trip. Here is a sample run:

```
Enter the driving distance: 900.5 ↵Enter
Enter miles per gallon: 25.5 ↵Enter
Enter price per gallon: 3.55 ↵Enter
The cost of driving is $125.36
```

Note

More than 200 additional programming exercises with solutions are provided to the instructors on the Instructor Resource Website.

CHAPTER 3

SELECTIONS

Objectives

- To declare `boolean` variables and write Boolean expressions using relational operators (§3.2).
- To implement selection control using one-way `if` statements (§3.3).
- To implement selection control using two-way `if-else` statements (§3.4).
- To implement selection control using nested `if` and multi-way `if` statements (§3.5).
- To avoid common errors and pitfalls in `if` statements (§3.6).
- To generate random numbers using the `Math.random()` method (§3.7).
- To program using selection statements for a variety of examples (`SubtractionQuiz`, `BMI`, `ComputeTax`) (§§3.7–3.9).
- To combine conditions using logical operators (`!`, `&&`, `||`, and `^`) (§3.10).
- To program using selection statements with combined conditions (`LeapYear`, `Lottery`) (§§3.11 and 3.12).
- To implement selection control using `switch` statements (§3.13).
- To write expressions using the conditional operator (§3.14).
- To examine the rules governing operator precedence and associativity (§3.15).
- To apply common techniques to debug errors (§3.16).