

starting out with >>>

JAVATM

EARLY OBJECTS

SIXTH EDITION



TONY GADDIS

starting out with >>>

JAVATM

EARLY OBJECTS

SIXTH EDITION



TONY GADDIS



Location of Videonotes in the Text

Compiling and Running a Java Program, p. [14](#)

[Chapter 1](#) Using an IDE, p. [14](#)

Your First Java Program, p. [28](#)

Displaying Console Output, p. [35](#)

Declaring Variables, p. [42](#)

[Chapter 2](#)

Simple Math Expressions, p. [58](#)

The Miles-per-Gallon Problem, p. [124](#)

Writing Classes and Creating Objects, p. [133](#)

[Chapter 3](#) Initializing an Object with a Constructor, p. [155](#)

The Personal Information Class Problem, p. [188](#)

The `if` Statement, p. [193](#)

The `if-else` Statement, p. [202](#)

[Chapter 4](#)

The `if-else-if` Statement, p. [217](#)

The Time Calculator Problem, p. [267](#)

The while Loop, p. [275](#)

[Chapter 5](#)

The Pennies for Pay Problem, p. [345](#)

Returning Objects from Methods, p. [385](#)

[Chapter 6](#)

Aggregation, p. [397](#)

The InventoryItem Class Copy Constructor Problem, p. [438](#)

Accessing Array Elements in a Loop, p. [450](#)

[Chapter 7](#)

Passing an Array to a Method, p. [468](#)

The Charge Account Validation Problem, p. [536](#)

[Chapter 8](#)

The Sentence Capitalizer Problem, p. [595](#)

Inheritance, p. [601](#)

[Chapter 9](#)

Polymorphism, p. [642](#)

The Employee and Production-Worker Classes Problem, p. [682](#)

Handling Exceptions, p. [687](#)

[Chapter 10](#)

The Exception Project Problem, p. [743](#)

Introduction to JavaFX, p. [748](#)

Creating Scenes, p. [751](#)

Displaying Images, p. [758](#)

The HBox Layout Container, p. [763](#)

The VBox Layout Container, p. [768](#)

The GridPane Layout Container, p. [770](#)

Chapter 11

Button Controls and Events, p. [778](#)

The TextField Control, p. [785](#)

Using Anonymous Inner Classes as Event Handlers, p. [789](#)

Using Lambda Expressions as Event Handlers, p. [792](#)

The Latin Translator Problem, p. [804](#)

JavaFX and CSS, p. [809](#)

RadioButton Controls, p. [824](#)

CheckBox Controls, p. [834](#)

Chapter 12 ListView Controls, p. [839](#)

ComboBox Controls, p. [860](#)

Slider Controls, p. [866](#)

The Dorm and Meal Plan Calculator Problem, p. [892](#)

Drawing Shapes with JavaFX, p. [895](#)

JavaFX Animation, p. [926](#)

JavaFX Effects, p. [944](#)

[Chapter 13](#) Playing Sound Files with JavaFX, p. [956](#)

Playing Videos with JavaFX, p. [960](#)

Handling Key Events in JavaFX, p. [965](#)

Handling Mouse Events in JavaFX, p. [972](#)

The Coin Toss Problem, p. [982](#)

Reducing a Problem with Recursion, p. [988](#)

[Chapter 14](#)

The Recursive Power Problem, p. [1010](#)

Connecting to a Database, p. [1016](#)

The SQL SELECT Statement, p. [1023](#)

Inserting Rows, p. [1044](#)

[Chapter 15](#) Updating and Deleting Rows, p. [1047](#)

Creating and Deleting Tables, p. [1056](#)

Relational Data, p. [1069](#)

The Customer Inserter Problem, p. [1092](#)

Starting Out with Java™

Early Objects

Sixth Edition

Starting Out with Java™

Early Objects

Sixth Edition

Tony Gaddis

Haywood Community College



330 Hudson Street, New York, NY 10013

Senior Vice President Courseware Portfolio Management:	Marcia J. Horton
Director, Portfolio Management: Engineering, Computer Science & Global Editions:	Julian Partridge
Portfolio Manager:	Matt Goldstein
Portfolio Management Assistant:	Kristy Alaura
Field Marketing Manager:	Demetrius Hall
Product Marketing Manager:	Yvonne Vannatta
Managing Producer, ECS and Math:	Scott Disanno
Content Producer:	Sandra L. Rodriguez
Composition:	iEnergizer Aptara [®] , Ltd.
Cover Designer:	Joyce Wells
Cover Photo:	Dimitris66/E+/Getty Images

Copyright © 2018, 2015, 2011, 2008, 2005 Pearson Education, Inc. Hoboken, NJ 07030. All rights reserved. Manufactured in the United States of America. This publication is protected by copyright and permissions should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit www.pearsoned.com/permissions/.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to

these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages with, or arising out of, the furnishing, performance, or use of these programs.

Pearson Education Ltd., London

Pearson Education Singapore, Pte. Ltd

Pearson Education Canada, Inc.

Pearson Education Japan

Pearson Education Australia PTY, Ltd

Pearson Education North Asia, Ltd., Hong Kong

Pearson Education de Mexico, S.A. de C.V.

Pearson Education Malaysia, Pte. Ltd.

Pearson Education, Inc., Hoboken

Library of Congress Cataloging-in-Publication Data

Names: Gaddis, Tony, author.

Title: Starting out with Java. Early objects / Tony Gaddis, Haywood Community College.

Description: Sixth edition. | Boston : Pearson Education, Inc., [2017] | Includes index.

Identifiers: LCCN 2016056455| ISBN 9780134462011 (alk. paper) | ISBN 0134462017 (alk. paper)

Subjects: LCSH: Java (Computer program language) | Object-oriented programming (Computer science)

Classification: LCC QA76.73.J38 G325 2017 | DDC 005.13/3—dc23 LC
record available at <https://lccn.loc.gov/2016056455>

1 17



ISBN-13: 978-0-13-446201-1

ISBN-10: 0-13-446201-7

Contents in Brief

1. [Preface xv](#)
1. [Chapter 1 Introduction to Computers and Java 1](#)
2. [Chapter 2 Java Fundamentals 29](#)
3. [Chapter 3 A First Look at Classes and Objects 129](#)
4. [Chapter 4 Decision Structures 193](#)
5. [Chapter 5 Loops and Files 275](#)
6. [Chapter 6 A Second Look at Classes and Objects 353](#)
7. [Chapter 7 Arrays and the ArrayList Class 445](#)
8. [Chapter 8 Text Processing and Wrapper Classes 543](#)
9. [Chapter 9 Inheritance 601](#)
10. [Chapter 10 Exceptions and Advanced File I/O 687](#)
11. [Chapter 11 JavaFX: GUI Programming and Basic Controls 745](#)
12. [Chapter 12 JavaFX: Advanced Controls 809](#)
13. [Chapter 13 JavaFX: Graphics, Effects, and Media 895](#)
14. [Chapter 14 Recursion 985](#)
15. [Chapter 15 Databases 1013](#)
 1. [Appendix A The ASCII/Unicode Characters 1095](#)
 2. [Appendix B Operator Precedence and Associativity 1097](#)

3. [Index 1099](#)
4. [Credits 1109](#)
5. Appendixes C–M Available on the book’s online resource page
6. Case Studies 1–4 Available on the book’s online resource page

Contents

1. [Preface xv](#)
1. [Chapter 1 Introduction to Computers and Java 1](#)
 1. [1.1 Introduction 1](#)
 2. [1.2 Why Program? 1](#)
 3. [1.3 Computer Systems: Hardware and Software 2](#)
 4. [1.4 Programming Languages 6](#)
 5. [1.5 What Is a Program Made of? 8](#)
 6. [1.6 The Programming Process 16](#)
 7. [1.7 Object-Oriented Programming 19](#)
 1. [Review Questions and Exercises 24](#)
 2. [Programming Challenge 28](#)
2. [Chapter 2 Java Fundamentals 29](#)
 1. [2.1 The Parts of a Java Program 29](#)
 2. [2.2 The `System.out.print` and `System.out.println` Methods, and the Java API 35](#)
 3. [2.3 Variables and Literals 41](#)
 4. [2.4 Primitive Data Types 47](#)
 5. [2.5 Arithmetic Operators 58](#)

6. [2.6 Combined Assignment Operators](#) 67
7. [2.7 Conversion between Primitive Data Types](#) 68
8. [2.8 Creating Named Constants with final](#) 72
9. [2.9 The String Class](#) 74
10. [2.10 Scope](#) 80
11. [2.11 Comments](#) 81
12. [2.12 Programming Style](#) 85
13. [2.13 Reading Keyboard Input](#) 87
14. [2.14 Dialog Boxes](#) 95
15. [2.15 Displaying Formatted Output with system.out.printf and String.format](#) 102
16. [2.16 Common Errors to Avoid](#) 116
 1. [Review Questions and Exercises](#) 118
 2. [Programming Challenges](#) 123
3. [Chapter 3 A First Look at Classes and Objects](#) 129
 1. [3.1 Classes](#) 129
 2. [3.2 More about Passing Arguments](#) 149
 3. [3.3 Instance Fields and Methods](#) 152
 4. [3.4 Constructors](#) 155
 5. [3.5 A BankAccount Class](#) 161
 6. [3.6 Classes, Variables, and Scope](#) 172

7. [3.7 Packages and `import` Statements 173](#)
8. [3.8 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities 174](#)
9. [3.9 Common Errors to Avoid 183](#)
 1. [Review Questions and Exercises 183](#)
 2. [Programming Challenges 187](#)
4. [Chapter 4 Decision Structures 193](#)
 1. [4.1 The `if` Statement 193](#)
 2. [4.2 The `if-else` Statement 202](#)
 3. [4.3 The `Payroll` Class 205](#)
 4. [4.4 Nested `if` Statements 209](#)
 5. [4.5 The `if-else-if` Statement 217](#)
 6. [4.6 Logical Operators 222](#)
 7. [4.7 Comparing `String` Objects 230](#)
 8. [4.8 More about Variable Declaration and Scope 235](#)
 9. [4.9 The Conditional Operator \(Optional\) 237](#)
 10. [4.10 The `switch` Statement 238](#)
 11. [4.11 Focus on Problem Solving: The `SalesCommission` Class 248](#)
 12. [4.12 Generating Random Numbers with the `Random` Class 255](#)
 13. [4.13 Common Errors to Avoid 261](#)
 1. [Review Questions and Exercises 262](#)

- 2. [Programming Challenges 267](#)
- 5. [Chapter 5 Loops and Files 275](#)
 - 1. [5.1 The Increment and Decrement Operators 275](#)
 - 2. [5.2 The `while` Loop 279](#)
 - 3. [5.3 Using the `while` Loop for Input Validation 286](#)
 - 4. [5.4 The `do-while` Loop 289](#)
 - 5. [5.5 The `for` Loop 292](#)
 - 6. [5.6 Running Totals and Sentinel Values 303](#)
 - 7. [5.7 Nested Loops 308](#)
 - 8. [5.8 The `break` and `continue` Statements 316](#)
 - 9. [5.9 Deciding Which Loop to Use 316](#)
 - 10. [5.10 Introduction to File Input and Output 317](#)
 - 11. [5.11 Common Errors to Avoid 337](#)
 - 1. [Review Questions and Exercises 338](#)
 - 2. [Programming Challenges 344](#)
- 6. [Chapter 6 A Second Look at Classes and Objects 353](#)
 - 1. [6.1 Static Class Members 353](#)
 - 2. [6.2 Overloaded Methods 360](#)
 - 3. [6.3 Overloaded Constructors 365](#)
 - 4. [6.4 Passing Objects as Arguments to Methods 372](#)

5. [6.5 Returning Objects from Methods 385](#)
 6. [6.6 The `toString` Method 388](#)
 7. [6.7 Writing an `equals` Method 392](#)
 8. [6.8 Methods That Copy Objects 394](#)
 9. [6.9 Aggregation 397](#)
 10. [6.10 The `this` Reference Variable 410](#)
 11. [6.11 Inner Classes 413](#)
 12. [6.12 Enumerated Types 416](#)
 13. [6.13 Garbage Collection 425](#)
 14. [6.14 Focus on Object-Oriented Design: Class Collaboration 427](#)
 15. [6.15 Common Errors to Avoid 431
 1. \[Review Questions and Exercises 432\]\(#\)
 2. \[Programming Challenges 437\]\(#\)](#)
7. [Chapter 7 Arrays and the `ArrayList` Class 445
 1. \[7.1 Introduction to Arrays 445\]\(#\)
 2. \[7.2 Processing Array Contents 456\]\(#\)
 3. \[7.3 Passing Arrays as Arguments to Methods 468\]\(#\)
 4. \[7.4 Some Useful Array Algorithms and Operations 472\]\(#\)
 5. \[7.5 Returning Arrays from Methods 484\]\(#\)
 6. \[7.6 String Arrays 486\]\(#\)](#)

7. [7.7 Arrays of Objects 490](#)
8. [7.8 The Sequential Search Algorithm 494](#)
9. [7.9 The Selection Sort and the Binary Search Algorithms 497](#)
10. [7.10 Two-Dimensional Arrays 505](#)
11. [7.11 Arrays with Three or More Dimensions 517](#)
12. [7.12 Command-Line Arguments and Variable-Length Argument Lists 518](#)
13. [7.13 The ArrayList Class 522](#)
14. [7.14 Common Errors to Avoid 530](#)
 1. [Review Questions and Exercises 531](#)
 2. [Programming Challenges 535](#)
8. [Chapter 8 Text Processing and Wrapper Classes 543](#)
 1. [8.1 Introduction to Wrapper Classes 543](#)
 2. [8.2 Character Testing and Conversion with the Character Class 544](#)
 3. [8.3 More about String Objects 551](#)
 4. [8.4 The StringBuilder Class 565](#)
 5. [8.5 Tokenizing Strings 574](#)
 6. [8.6 Wrapper Classes for the Numeric Data Types 583](#)
 7. [8.7 Focus on Problem Solving: The TestScoreReader Class 587](#)
 8. [8.8 Common Errors to Avoid 591](#)

1. [Review Questions and Exercises 591](#)
2. [Programming Challenges 595](#)
9. [Chapter 9 Inheritance601](#)
 1. [9.1 What Is Inheritance? 601](#)
 2. [9.2 Calling the Superclass Constructor 612](#)
 3. [9.3 Overriding Superclass Methods 620](#)
 4. [9.4 Protected Members 628](#)
 5. [9.5 Classes That Inherit from Subclasses 635](#)
 6. [9.6 The `Object` Class 640](#)
 7. [9.7 Polymorphism 642](#)
 8. [9.8 Abstract Classes and Abstract Methods 647](#)
 9. [9.9 Interfaces 653](#)
 10. [9.10 Anonymous Inner Classes 668](#)
 11. [9.11 Functional Interfaces and Lambda Expressions 670](#)
 12. [9.12 Common Errors to Avoid 675](#)
 1. [Review Questions and Exercises 676](#)
 2. [Programming Challenges 682](#)
10. [Chapter 10 Exceptions and Advanced File I/O 687](#)
 1. [10.1 Handling Exceptions 687](#)
 2. [10.2 Throwing Exceptions 710](#)

- 3. [10.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization](#) 718
 - 4. [10.4 Common Errors to Avoid](#) 734
 - 1. [Review Questions and Exercises](#) 735
 - 2. [Programming Challenges](#) 741
11. [Chapter 11 JavaFX: GUI Programming and Basic Controls](#) 745
- 1. [11.1 Graphical User Interfaces](#) 745
 - 2. [11.2 Introduction to JavaFX](#) 748
 - 3. [11.3 Creating Scenes](#) 751
 - 4. [11.4 Displaying Images](#) 758
 - 5. [11.5 More about the HBox, VBox, and GridPane Layout Containers](#) 762
 - 6. [11.6 Button Controls and Events](#) 778
 - 7. [11.7 Reading Input with TextField Controls](#) 785
 - 8. [11.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events](#) 789
 - 9. [11.9 The BorderPane Layout Container](#) 794
 - 10. [11.10 The ObservableList Interface](#) 799
 - 11. [11.11 Common Errors to Avoid](#) 800
- 1. [Review Questions and Exercises](#) 800
 - 2. [Programming Challenges](#) 804

- 12. [Chapter 12 JavaFX: Advanced Controls 809](#)
 - 1. [12.1 Styling JavaFX Applications with CSS 809](#)
 - 2. [12.2 RadioButton Controls 824](#)
 - 3. [12.3 CheckBox Controls 834](#)
 - 4. [12.4 ListView Controls 839](#)
 - 5. [12.5 comboBox Controls 860](#)
 - 6. [12.6 slider Controls 866](#)
 - 7. [12.7 TextArea Controls 871](#)
 - 8. [12.8 Menus 873](#)
 - 9. [12.9 The FileChooser Class 883](#)
 - 10. [12.10 Using Console Output to Debug a GUI Application 884](#)
 - 11. [12.11 Common Errors to Avoid 888](#)
 - 1. [Review Questions 888](#)
 - 2. [Programming Challenges 892](#)
- 13. [Chapter 13 JavaFX: Graphics, Effects, and Media 895](#)
 - 1. [13.1 Drawing Shapes 895](#)
 - 2. [13.2 Animation 926](#)
 - 3. [13.3 Effects 944](#)
 - 4. [13.4 Playing Sound Files 955](#)
 - 5. [13.5 Playing Videos 960](#)

6. [13.6 Handling Key Events 965](#)
7. [13.7 Handling Mouse Events 972](#)
8. [13.8 Common Errors to Avoid 978](#)
 1. [Review Questions 978](#)
 2. [Programming Challenges 981](#)
14. [Chapter 14 Recursion 985](#)
 1. [14.1 Introduction to Recursion 985](#)
 2. [14.2 Solving Problems with Recursion 988](#)
 3. [14.3 Examples of Recursive Methods 992](#)
 4. [14.4 A Recursive Binary Search Method 999](#)
 5. [14.5 The Towers of Hanoi 1002](#)
 6. [14.6 Common Errors to Avoid 1006](#)
 1. [Review Questions and Exercises 1007](#)
 2. [Programming Challenges 1009](#)
15. [Chapter 15 Databases 1013](#)
 1. [15.1 Introduction to Database Management Systems 1013](#)
 2. [15.2 Tables, Rows, and Columns 1019](#)
 3. [15.3 Introduction to the SQL SELECT Statement 1023](#)
 4. [15.4 Inserting Rows 1044](#)
 5. [15.5 Updating and Deleting Existing Rows 1047](#)

6. [15.6 Creating and Deleting Tables 1056](#)
7. [15.7 Creating a New Database with JDBC 1060](#)
8. [15.8 Scrollable Result Sets 1061](#)
9. [15.9 Result Set Metadata 1063](#)
10. [15.10 Relational Data 1069](#)
11. [15.11 Advanced Topics 1084](#)
12. [15.12 Common Errors to Avoid 1086](#)
 1. [Review Questions and Exercises 1087](#)
 2. [Programming Challenges 1092](#)
1. [Appendix A The ASCII/Unicode Characters 1095](#)
2. [Appendix B Operator Precedence and Associativity 1097](#)
3. [Index 1099](#)
4. [Credits 1109](#)
5. **Available on the Computer Science Portal at
www.pearsonhighered.com/gaddis:**
6. Appendix C Java Key Words
7. Appendix D Installing the JDK and Using the JDK Documentation
8. Appendix E Using the javadoc Utility
9. Appendix F More about the Math Class
10. Appendix G Packages
11. Appendix H Working with Records and Random-Access Files

12. Appendix I Configuring Java DB
13. Appendix J The QuickSort Algorithm
14. Appendix K Named Colors
15. Appendix L Answers to Checkpoints Questions
16. Appendix M Answers to Odd-Numbered Review Questions
17. Case Study 1 The Amortization Class
18. Case Study 2 The PinTester Class
19. Case Study 3 Parallel Arrays
20. Case Study 4 The SerialNumber Class

Preface

Welcome to *Starting Out with Java: Early Objects*, Sixth Edition. This book is intended for a one-semester or a two-quarter CS1 course. Although it is written for students with no prior programming background, even experienced students will benefit from its depth of detail.

Early Objects, Late Graphics

The approach taken by this text can be described as “early objects, late graphics.” The student is introduced to object-oriented programming (OOP) early in the book. The fundamentals of control structures, classes, and the OOP paradigm are thoroughly covered before moving on to graphics and more powerful applications of the Java language.

As with all the books in the *Starting Out With* series, the hallmark of this text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in example programs that are concise and practical.

Changes in the Sixth Edition

The most significant change in this edition is the switch from Swing to JavaFX in the chapters that focus on GUI development. Although Swing is not officially deprecated, Oracle has announced that JavaFX is replacing Swing as the standard GUI library for Java¹.

¹ <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>

The Swing and Applet material that appeared in the previous edition is still available on the book’s Website as the following online chapters:

- The previous [Chapter 11 GUI Applications—Part 1](#) is now available online as Chapter 16.
- The previous [Chapter 12 GUI Applications—Part 2](#) is now available online as Chapter 17.
- The previous [Chapter 13 Applets and More](#) is now available online as Chapter 18.

In this edition, we have added the following new chapters:

- [Chapter 11 JavaFX: GUI Programming and Basic Controls](#)

This chapter presents the basics of developing graphical user interface (GUI) applications with JavaFX. Fundamental controls, layout containers, and the basic concepts of event-driven programming are covered.

- [Chapter 12 JavaFX: Advanced Controls](#)

This chapter discusses CSS styling and advanced user interface controls.

- [Chapter 13 JavaFX: Graphics, Effects, and Media](#)

This chapter discusses 2D shapes, animation, visual effects, playing audio and video, and responding to mouse and keyboard events.



Note:

[Chapter 14](#) from the previous edition has also been moved to the book's Website as *Chapter 19 Creating JavaFX Applications with Scene Builder*. Although Oracle no longer officially supports Scene Builder, it is still available as an open source tool at <http://gluonhq.com/labs/scene-builder/>

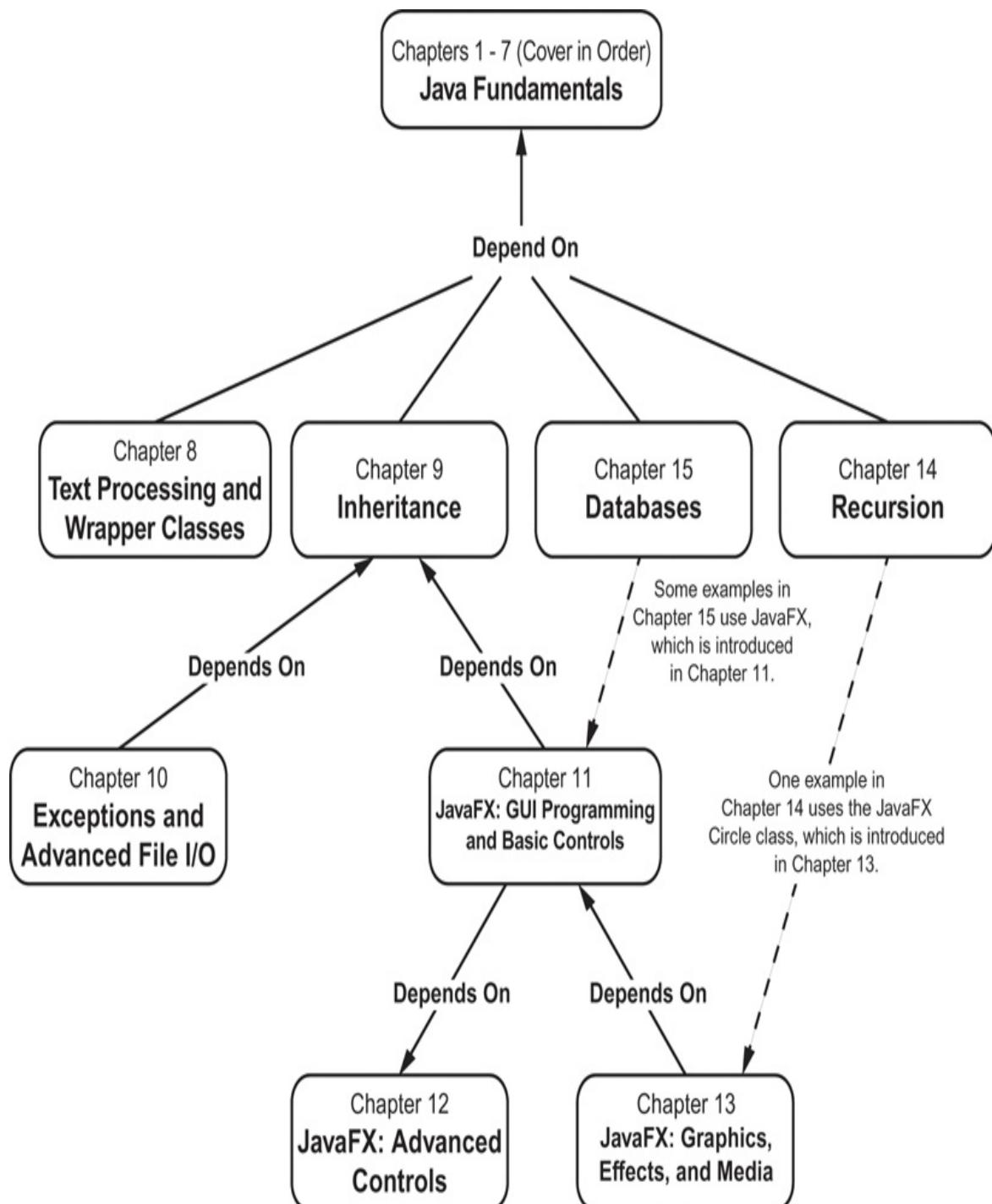
In addition to the new JavaFX chapters, the following updates were made in this edition:

- **String.format Is Used Instead of DecimalFormat:** In previous editions, the `DecimalFormat` class was used to format strings for GUI output. In this edition, the `String.format` method (which is introduced in [Chapter 2](#)) is used instead. With `String.format`, the student can use the same format specifiers and flags that were learned with the `System.out.printf` method.
- **StringTokenizer Is No Longer Used:** In previous editions, the `StringTokenizer` class was introduced as a way to tokenize strings. In this edition, all string tokenizing is done with the `String.split` method.
- **Introduction of @Override annotation:** [Chapter 9](#) now introduces the use of `@Override` annotation, and explains how it can prevent subtle errors.
- **A New Section on Anonymous Inner Classes:** [Chapter 9](#) now has a new section that introduces anonymous inner classes.
- **The Introduction to Interfaces Has Been Improved:** The introductory material on interfaces in [Chapter 9](#) has been revised for greater clarity.
- **Default Methods:** In this edition, [Chapter 9](#) provides new material on default methods in interfaces, a feature that was introduced in Java 8.
- **Functional Interfaces and Lambda Expressions:** Java 8 introduced functional interfaces and lambda expressions, and in this edition, [Chapter 9](#) has a new section on these topics. The new material gives a detailed, stepped-out explanation of lambda expressions, and discusses how they can be used to instantiate objects of anonymous classes that implement functional interfaces.
- **JavaFX in the Database Chapter:** The Database chapter, which is now [Chapter 15](#), has been updated to use JavaFX instead of Swing for its GUI applications.
- **New Programming Problems:** Several new, motivational programming problems have been added throughout the book.

Organization of the Text

The text teaches Java step-by-step. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, there is some flexibility. [Figure P-1](#) shows chapter dependencies. Each box represents a chapter or a group of chapters. A solid-line arrow points from one chapter to the chapter that must be covered previously. A dotted-line arrow indicates that only a section or minor portion of the chapter depends on another chapter.

Figure P-1 Chapter dependencies



[Figure P-1 Full Alternative Text](#)

Brief Overview of Each Chapter

[Chapter 1](#): Introduction to Computers and Java. This chapter provides an introduction to the field of computer science, and covers the fundamentals of hardware, software, and programming languages. The elements of a program, such as key words, variables, operators, and punctuation are discussed through the examination of a simple program. An overview of entering source code, compiling it, and executing it is presented. A brief history of Java is also given. The chapter concludes with a primer on OOP.

[Chapter 2](#): Java Fundamentals. This chapter gets the student started in Java by introducing data types, identifiers, variable declarations, constants, comments, program output, and arithmetic operations. The conventions of programming style are also introduced. The student learns to read console input with the `Scanner` class, or as an option, through dialog boxes with `JOptionPane`. Formatting output with the `System.out.printf` and `String.format` methods is covered.

[Chapter 3](#): A First Look at Classes and Objects. This chapter introduces the student to classes. Once the student learns about fields and methods, UML diagrams are introduced as a design tool. The student learns to write simple `void` methods, as well as simple methods that return a value. Arguments and parameters are also discussed. Finally, the student learns how to write constructors, and the concept of the default constructor is discussed. A `BankAccount` class is presented as a case study, and a section on object-oriented design is included. This section leads the students through the process of identifying classes and their responsibilities within a problem domain. There is also a section that briefly explains packages and the `import` statement.

[Chapter 4](#): Decision Structures. Here the student explores relational operators and relational expressions and is shown how to control the flow of a program with the `if`, `if/else`, and `if/else if` statements. The conditional operator and the `switch` statement are also covered. This chapter also discusses how to

compare `String` objects with the `equals`, `compareTo`, `equalsIgnoreCase`, and `compareToIgnoreCase` methods. An object-oriented case study shows how lengthy algorithms can be decomposed into several methods.

[Chapter 5](#): Loops and Files. This chapter covers Java's repetition control structures. The `while` loop, `do-while` loop, and `for` loop are taught, along with common uses for these devices. Counters, accumulators, running totals, sentinels, and other application-related topics are discussed. Simple file operations for reading and writing text files are also covered.

[Chapter 6](#): A Second Look at Classes and Objects. This chapter shows students how to write classes with added capabilities. Static methods and fields, interaction between objects, passing objects as arguments, and returning objects from methods are discussed. Aggregation and the "has a" relationship is covered, as well as enumerated types. A section on object-oriented design shows how to use CRC (class, responsibilities, and collaborations) cards to determine the collaborations among classes.

[Chapter 7](#): Arrays and the `ArrayList` Class. In this chapter, students learn to create and work with single and multidimensional arrays. Numerous array-processing techniques are demonstrated, such as summing the elements in an array, finding the highest and lowest values, and sequentially searching an array are also discussed. Other topics, including ragged arrays and variable-length arguments (`varargs`), are also discussed. The `ArrayList` class is introduced, and Java's generic types are briefly discussed and demonstrated.

[Chapter 8](#): Text Processing and Wrapper Classes. This chapter discusses the numeric and character wrapper classes. Methods for converting numbers to strings, testing the case of characters, and converting the case of characters are covered. Autoboxing and unboxing are also discussed. More `String` class methods are covered, including using the `split` method to tokenize strings. The chapter also covers the `StringBuilder` class and the `String` class's `split` method.

[Chapter 9](#): Inheritance. The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include superclass and subclass constructors, method overriding, polymorphism and dynamic binding, protected and package access, class hierarchies, abstract

classes and methods, anonymous inner classes, interfaces, and lambda expressions.

[Chapter 10](#): Exceptions and Advanced File I/O. In this chapter, the student learns to develop enhanced error trapping techniques using exceptions. Handling an exception is covered, as well as developing and throwing custom exceptions. This chapter also discusses advanced techniques for working with sequential access, random access, text, and binary files.

[Chapter 11](#): JavaFX: GUI Programming and Basic Controls. This chapter presents the basics of developing graphical user interface (GUI) applications with JavaFX. Fundamental controls, layout containers, and the basic concepts of event-driven programming are covered.

[Chapter 12](#): JavaFX: Advanced Controls. This chapter discusses CSS styling and advanced user interface controls, such as RadioButtons, CheckBoxes, ListViews, ComboBoxes, Sliders, and TextAreas. Menu systems and FileChoosers are also covered.

[Chapter 13](#): JavaFX: Graphics, Effects, and Media. This chapter discusses 2D shapes, animation, visual effects, playing audio and video, and responding to mouse and keyboard events.

[Chapter 14](#): Recursion. This chapter presents recursion as a problem-solving technique. Numerous examples of recursion are demonstrated.

[Chapter 15](#): Databases. This chapter introduces the student to database programming. The basic concepts of database management systems and SQL are first presented. Then the student learns to use JDBC to write database applications in Java. Relational data is covered, and numerous example programs are presented throughout the chapter.

[Appendix A](#). The ASCII/Unicode Characters

[Appendix B](#). Operator Precedence and Associativity

The following appendices, online chapters, and online case studies are available at www.pearsonhighered.com/gaddis.

Online Appendices

Appendix C: Java Key Words

Appendix D: Installing the JDK and Using the JDK Documentation

Appendix E: Using the `javadoc` Utility

Appendix F: More About the `Math` Class

Appendix G: Packages

Appendix H: Working with Records and Random-Access Files

Appendix I: Configuring JavaDB

Appendix J: The QuickSort Algorithm

Appendix K: Named Colors

Appendix L: Answers to Checkpoint Questions

Appendix M: Answers to Odd-Numbered Review Questions

Online Chapters

Chapter 16: GUI Applications with Swing—Part 1

Chapter 17: GUI Applications with Swing—Part 2

Chapter 18: Applets and More

Chapter 19: Creating JavaFX Applications with Scene Builder

Online Case Studies

Case Study 1: The Amortization Class

Case Study 2: The PinTester Class

Case Study 3: Parallel Arrays

Case Study 4: The SerialNumber Class

Features of the Text

Concept Statements

Each major section of the text starts with a concept statement. This statement summarizes the ideas of the section.

Example Programs

The text has an abundant number of complete example programs, each designed to highlight the topic currently being studied. In most cases, these are practical, real-world examples. Source code for these programs is provided so that students can run the programs themselves.

Program Output

After each example program, there is a sample of its screen output. This immediately shows the student how the program should function.



Checkpoints

Checkpoints are questions placed throughout each chapter as a self-test study

aid. Answers for all Checkpoint questions are found in Appendix L (available for download) so students can check how well they have learned a new topic. To download Appendix L, go to the Gaddis resource page at www.pearsonhighered.com/gaddis.



Note:

Notes appear at appropriate places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.



Warning!

Warnings are notes that caution the student about certain Java features, programming techniques, or practices that can lead to malfunctioning programs or lost data.



VideoNotes

A series of online videos, developed specifically for this book, are available for viewing at www.pearsonhighered.com/gaddis. Icons appear throughout the text alerting the student to videos about specific topics.

Case Studies

Case studies that simulate real-world applications appear in many chapters throughout the text, with complete code provided for each. These case studies are designed to highlight the major topics of the chapter in which they appear.

Review Questions and Exercises

Each chapter presents a thorough and diverse set of review questions and exercises. They include Multiple Choice and True/False, Find the Error, Algorithm Workbench, and Short Answer.

Programming Challenges

Each chapter offers a pool of programming challenges designed to solidify students' knowledge of topics at hand. In most cases, the assignments present real-world problems to be solved.



In the Spotlight

Many of the chapters provide an *In the Spotlight* section that presents a programming problem, along with detailed, step-by-step analysis showing the student how to solve it.

Supplements

Companion Website

Many student resources are available for this book from the Computer Science Portal. The following items are available at www.pearsonhighered.com/gaddis using the Access Code found on the inside front cover of the book:

- The source code for each example program in the book
- Access to the book's VideoNotes

- Online Chapters 16 through 19
- Appendixes C–M (listed in the Table of Contents)
- A collection of four valuable Case Studies (listed in the Table of Contents)
- Links to download the JavaTM Development Kit
- Links to download numerous programming environments, including jGRASPTM, EclipseTM, TextPadTM, NetBeansTM, JCreator, and DrJava

Online Practice and Assessment with 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.

Instructor Resources

The following supplements are available to qualified instructors only. Visit the Pearson Education Instructor Resource Center (www.pearsonhighered.com/irc) for information on how to access them:

- Answers to all Review Questions in the text
- Solutions for all Programming Challenges in the text
- PowerPoint presentation slides for every chapter
- Computerized test bank

Acknowledgments

There have been many helping hands in the development and publication of this text. I would like to thank the following faculty reviewers for their helpful suggestions and expertise during the production of this text:

Dan Dao
Richland College

Edward Souza
Hawai'i Pacific University

Kurt Kominek
Northeast State Community College

Rebecca Caldwell
Winston-Salem State University

Joram Ngwenya
North Carolina Central University

Renae Bagzis

Southwestern Oklahoma State University

Tyler Baysinger
River Valley Community College

James Belton
Winston-Salem State University

Jim Caprio
Niagara County Community College

Reviewers For Previous Editions

Ahmad Abuhejleh
University of Wisconsin–River Falls

Colin Archibald
Valencia CC

Ijaz Awani
Savannah State University

Dr. Charles W. Bane
Tarleton State University

Dwight Barnett
Virginia Tech

Asoke Bhattacharyya
Saint Xavier University, Chicago

Marvin Bishop
Manhattan College

Heather Booth
University Tennessee—Knoxville

David Boyd
Valdosta University

Julius Brandstatter
Golden Gate University

Rebecca Caldwell
Winston-Salem State University

Kim Cannon
Greenville Tech

James Chegwidden
Tarrant County College

Kay Chen
Bucks County Community College

Brad Chilton
Tarleton State University

Diane Christie
University of Wisconsin–Stout

Cara Cocking
Marquette University

Walter C. Daugherty
Texas A&M University

Dan Dao
Richland College

Michael Doherty
University of the Pacific

Jeanne M. Douglas
University of Vermont

Sander Eller
California Polytechnic University—Pomona

Brooke Estabrook-Fishinghawk
Mesa Community College

Mike Fry
Lebanon Valley College

Georgia R. Grant
College of San Mateo

Chris Haynes
Indiana University

Ric Heishman
Northern Virginia Community College

Naser Heravi
College of Southern Nevada

Deedee Herrera
Dodge City Community College

Mary Hovik
Lehigh Carbon Community College

Brian Howard
DePauw University

Deborah Hughes
Lyndon State College

Norm Jacobson
University of California at Irvine

Nicole Jiao
South Texas College

Dr. Stephen Judd
University of Pennsylvania

Kurt Kominek
Northeast State Community College

Harry Lichtbach
Evergreen Valley College

Michael A. Long
California State University, Chico

Tim Margush
University of Akron

Blayne E. Mayfield
Oklahoma State University

Scott McLeod
Riverside Community College

Dean Mellas
Cerritos College

Kevin Mess
College of Southern Nevada

Georges Merx
San Diego Mesa College

Martin Meyers
California State University, Sacramento

Pati Milligan
Baylor University

Godfrey Muganda
North Central College

Steve Newberry
Tarleton State University

Lynne O'Hanlon
Los Angeles Pierce College

Lisa Olivieri
Chestnut Hill College

Merrill Parker
Chattanooga State Technical Community College

Bryson R. Payne
North Georgia College and State University

Rodney Pearson
Mississippi State University

Peter John Polito
Springfield College

Charles Robert Putnam
California State University, Northridge
Dr. Y. B. Reddy
Grambling State University

Carolyn Schauble
Colorado State University

Bonnie Smith
Fresno City College

Daniel Spiegel
Kutztown University

Caroline St. Clair
North Central College

Karen Stanton

Los Medanos College

Mark Swanson
Southeast Technical

Peter H. Van Der Goes
Rose State College

Tuan A Vo
Mt. San Antonio College

Xiaoying Wang
University of Mississippi

I would also like to thank my family and friends for their support in all of my projects. I am extremely fortunate to have Matt Goldstein as my editor, and Kristy Alaura as editorial assistant. Their guidance and encouragement make it a pleasure to write chapters and meet deadlines. I am also fortunate to have Demetrius Hall as my marketing manager. His hard work is truly inspiring, and he does a great job of getting this book out to the academic community. The production team, led by Sandra Rodriguez, worked tirelessly to make this book a reality. Thanks to you all!

About the Author

Tony Gaddis is the principal author of the *Starting Out With* series of textbooks. Tony has nearly twenty years experience teaching computer science courses at Haywood Community College in North Carolina. He is a highly acclaimed instructor who was previously selected as the North Carolina Community College Teacher of the Year and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. Besides Java™ books, the Starting Out series includes introductory books using the C++ programming language, Microsoft® Visual Basic®, Microsoft® C#®, Python, Programming Logic and Design, Alice, and App Inventor, all published by Pearson.

BREAK THROUGH
To improving results

MyProgrammingLab™

Through the power of practice and immediate personalized feedback,
MyProgrammingLab helps improve your students' performance.

PROGRAMMING PRACTICE

With MyProgrammingLab, your students will gain first-hand programming experience in an interactive online environment.

IMMEDIATE, PERSONALIZED FEEDBACK

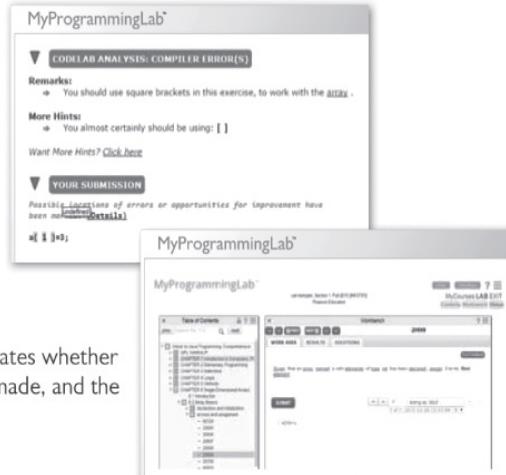
MyProgrammingLab automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.

GRADUATED COMPLEXITY

MyProgrammingLab breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.



PEARSON eTEXT

The Pearson eText gives students access to their textbook anytime, anywhere.

STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyProgrammingLab**,
please visit www.myprogramminglab.com.

Copyright © 2018 Pearson Education, Inc. or its affiliate(s). All rights reserved. HELO88173 • 11/15

ALWAYS LEARNING

PEARSON

Chapter 1 Introduction to Computers and Java

Topics

1. [1.1 Introduction](#)
2. [1.2 Why Program?](#)
3. [1.3 Computer Systems: Hardware and Software](#)
4. [1.4 Programming Languages](#)
5. [1.5 What Is a Program Made of?](#)
6. [1.6 The Programming Process](#)
7. [1.7 Object-Oriented Programming](#)

1.1 Introduction

This book teaches programming using Java. Java is a powerful language that runs on practically every type of computer. It can be used to create large applications, small programs, mobile applications, and code that powers a website. Before plunging right into learning Java, however, this chapter will review the fundamentals of computer hardware and software, then take a broad look at computer programming in general.

1.2 Why Program?

Concept:

Computers can do many different jobs because they are programmable.

Every profession has tools that make the job easier to do. Carpenters use hammers, saws, and measuring tapes. Mechanics use wrenches, screwdrivers, and ratchets. Electronics technicians use probes, scopes, and meters. Some tools are unique and can be categorized as belonging to a single profession. For example, surgeons have certain tools that are designed specifically for surgical operations. Those tools probably aren't used by anyone other than surgeons. There are some tools, however, that are used in several professions. Screwdrivers, for instance, are used by mechanics, carpenters, and many others.

The computer is a tool used by so many professions that it cannot be easily categorized. It can perform so many different tasks that it is perhaps the most versatile tool ever made. To the accountant, computers balance books, analyze profits and losses, and prepare tax reports. To the factory worker, computers control manufacturing machines and track production. To the mechanic, computers analyze the various systems in an automobile and pinpoint hard-to-find problems. The computer can do such a wide variety of tasks because it can be *programmed*. It is a machine specifically designed to follow instructions. Because of the computer's programmability, it doesn't belong to any single profession. Computers are designed to do whatever job their programs, or *software*, tell them to do.

Computer programmers do a very important job. They create software that transforms computers into the specialized tools of many trades. Without programmers, the users of computers would have no software, and without software, computers would not be able to do anything.

Computer programming is both an art and a science. It is an art because every aspect of a program should be carefully designed. Here are a few of the things that must be designed for any real-world computer program:

- The logical flow of the instructions
- The mathematical procedures
- The layout of the programming statements
- The appearance of the screens
- The way information is presented to the user
- The program's "user friendliness"
- Help systems and written documentation

There is also a science to programming. Because programs rarely work right the first time they are written, a lot of analyzing, experimenting, correcting, and redesigning is required. This demands patience and persistence of the programmer. Writing software demands discipline as well. Programmers must learn special languages such as Java because computers do not understand English or other human languages. Programming languages have strict rules that must be carefully followed.

Both the artistic and scientific nature of programming make writing computer software like designing a car: Both cars and programs should be functional, efficient, powerful, easy to use, and visually appealing.

1.3 Computer Systems: Hardware and Software

Concept:

All computer systems consist of similar hardware devices and software components.

Hardware

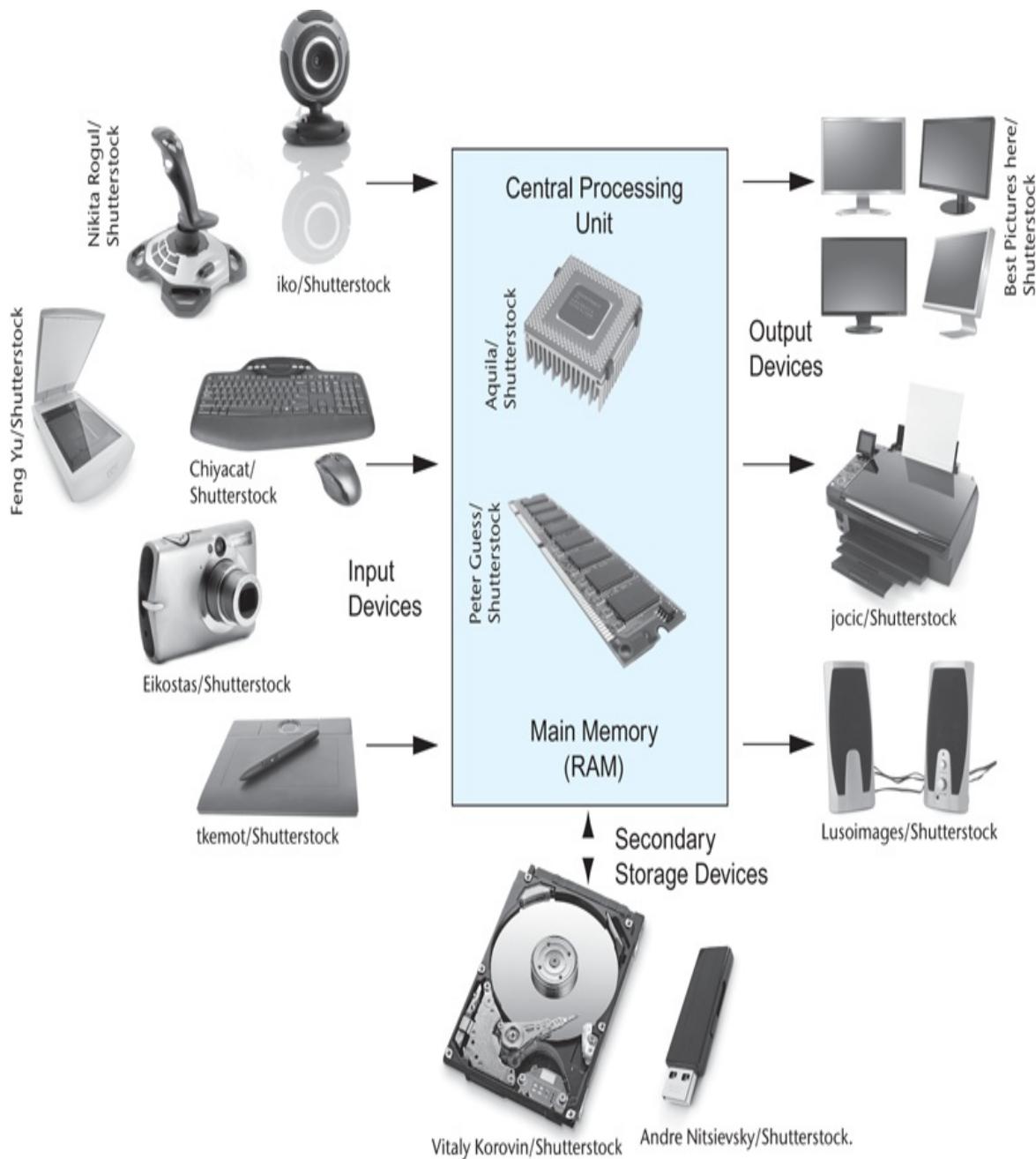
Hardware refers to the physical components of which a computer is made. A computer, as we generally think of it, is not an individual device, but a system of devices. Like the instruments in a symphony orchestra, each device plays its own part. A typical computer system consists of the following major components:

- The central processing unit
- Main memory
- Secondary storage devices
- Input devices
- Output devices

The organization of a computer system is shown in [Figure 1-1](#).

Figure 1-1 The organization of

a computer system



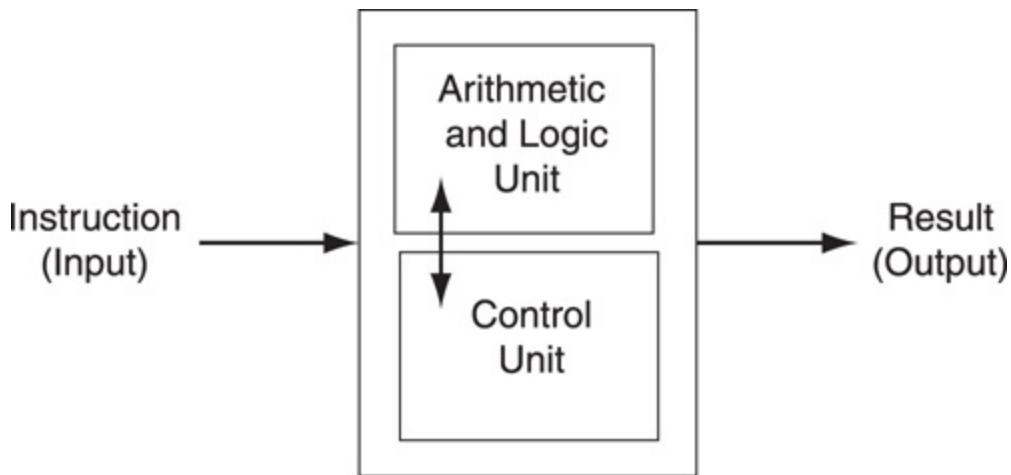
[Figure 1-1 Full Alternative Text](#)

Let's take a closer look at each of these devices.

The CPU

At the heart of a computer is its *central processing unit*, or *CPU*. The CPU's job is to fetch instructions, follow the instructions, and produce some resulting data. Internally, the central processing unit consists of two parts: the *control unit*, and the *arithmetic and logic unit (ALU)*. The control unit coordinates all of the computer's operations. It is responsible for determining where to get the next instruction, and for regulating the other major components of the computer with control signals. The arithmetic and logic unit, as its name suggests, is designed to perform mathematical operations. The organization of the CPU is shown in [Figure 1-2](#).

Figure 1-2 The organization of the CPU



[Figure 1-2 Full Alternative Text](#)

A program is a sequence of instructions stored in the computer's memory. When a computer is running a program, the CPU is engaged in a process formally known as the *fetch/decode/execute cycle*. The steps in the fetch/decode/execute cycle are as follows:

- Fetch* The CPU's control unit fetches, from main memory, the next instruction in the sequence of program instructions.
- Decode* The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal. The signal is routed to the appropriate component of the computer
- Execute* (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation.

These steps are repeated as long as there are instructions to perform.

Main Memory

Commonly known as *random-access memory*, or *RAM*, the computer's main memory is a device that holds data. Specifically, RAM holds the sequences of instructions in the programs that are running, and the data those programs are using.

Memory is divided into sections that hold an equal amount of data. Each section is made of eight "switches" that may be either on or off. A switch in the on position usually represents the number 1, although a switch in the off position usually represents the number 0. The computer stores data by setting the switches in a memory location to a pattern that represents a character or a number. Each of these switches is known as a *bit*, which stands for *binary digit*. Each section of memory, which is a collection of eight bits, is known as a *byte*. Each byte is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A byte is identified by its address in much the same way a post office box is identified by an address. [Figure 1-3](#) shows a series of bytes with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the byte at address 16, and the number 72 is stored in the byte at address 23.

Figure 1-3 Memory bytes and their addresses

0	1	2	3	4	5	6	7	8	9	
10	11	12	13	14	15	16	149	17	18	19
20	21	22	23	72	24	25	26	27	28	29

RAM is usually a volatile type of memory, used only for temporary storage. When the computer is turned off, the contents of RAM are erased.

Secondary Storage

Secondary storage is a type of memory that can hold data for long periods of time—even when there is no power to the computer. Programs are normally stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory figures, is saved to secondary storage as well.

The most common type of secondary storage device is the *disk drive*. A traditional disk drive stores data by magnetically encoding it onto a spinning circular disk. *Solid-state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts and operates faster than a traditional disk drive. Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External drives are also available that connect to one of the computer’s communication ports. External drives can be used to create backup copies of important data, or to move data to another computer.

In addition to external drives, many types of devices have been created for copying data and for moving it to other computers. *Universal Serial Bus drives*, or *USB drives*, are small devices that plug into the computer’s USB port and appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also used for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they make a good medium for creating backup copies of data.

Input Devices

Input is any data the computer collects from the outside world. The device that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, microphone, webcam, and digital camera. Disk drives, optical drives, and USB drives can also be considered input devices, because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer sends to the outside world. It might be a sales report, a list of names, or a graphic image. The data is sent to an output device, which formats and presents it. Common output devices are monitors and printers. Disk drives, USB drives, and CD/DVD recorders can also be considered output devices, because the CPU sends data to them in order to be saved.

Software

As previously mentioned, software refers to the programs that run on a computer. There are two general categories of software: operating systems, and application software. An *operating system* is a set of programs that manages the computer's hardware devices and controls their processes. Most modern operating systems are multitasking, which means they are capable of

running multiple programs at once. Through a technique called *time sharing*, a multitasking system divides the allocation of hardware resources and the attention of the CPU among all the executing programs. UNIX, Linux, and modern versions of Windows and Mac OS are multitasking operating systems.

Application software refers to programs that make the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user. Word processing, spreadsheet, and database programs are all examples of application software.



Checkpoint

1. 1.1 Why is the computer used by so many different people, in so many different professions?
2. 1.2 List the five major hardware components of a computer system.
3. 1.3 Internally, the CPU consists of what two units?
4. 1.4 Describe the steps in the fetch/decode/execute cycle.
5. 1.5 What is a memory address? What is its purpose?
6. 1.6 Explain why computers have both main memory and secondary storage.
7. 1.7 What does the term “multitasking” mean?

1.4 Programming Languages

Concept:

A program is a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.

What Is a Program?

Computers are designed to follow instructions. A computer program is a set of instructions that enable the computer to solve a problem or perform a task. For example, suppose we want the computer to calculate someone's gross pay. The following is a list of things the computer should do to perform this task.

1. Display a message on the screen: "How many hours did you work?"
2. Allow the user to enter the number of hours worked.
3. Once the user enters a number, store it in memory.
4. Display a message on the screen: "How much do you get paid per hour?"
5. Allow the user to enter an hourly pay rate.
6. Once the user enters a number, store it in memory.
7. Once both the number of hours worked and the hourly pay rate are entered, multiply the two numbers and store the result in memory.
8. Display a message on the screen that shows the amount of money

earned. The message must include the result of the calculation performed in Step 7.

Collectively, these instructions are called an *algorithm*. An algorithm is a set of well-defined steps for performing a task or solving a problem. Notice that these steps are sequentially ordered. Step 1 should be performed before Step 2, and so forth. It is important that these instructions be performed in their proper sequence.

Although you and I might easily understand the instructions in the pay-calculating algorithm, it is not ready to be executed on a computer. A computer's CPU can only process instructions that are written in machine language. If you were to look at a machine language program, you would see a stream of binary numbers (numbers consisting of only 1s and 0s). The binary numbers form machine language instructions, which the CPU interprets as commands. Here is an example of what a machine language instruction might look like:

1011010000000101

As you can imagine, the process of encoding an algorithm in machine language is very tedious and difficult. In addition, each different type of CPU has its own machine language. If you wrote a machine language program for computer A and then wanted to run it on computer B, which has a different type of CPU, you would have to rewrite the program in computer B's machine language.

Programming languages, which use words instead of numbers, were invented to ease the task of programming. A program can be written in a programming language, which is much easier to understand than machine language, and then translated into machine language. Programmers use software to perform this translation. Many programming languages have been created. [Table 1-1](#) lists a few of the well-known ones.

Table 1-1 Programming languages

Language	Description
BASIC	Beginners All-purpose Symbolic Instruction Code is a general-purpose, procedural programming language. It was originally designed to be simple enough for beginners to learn.
FORTRAN	FORmula TRANslator is a procedural language designed for programming complex mathematical algorithms.
COBOL	Common Business-Oriented Language is a procedural language designed for business applications.
Pascal	Pascal is a structured, general-purpose, procedural language designed primarily for teaching programming.
C	C is a structured, general-purpose, procedural language developed at Bell Laboratories.
C++	Based on the C language, C++ offers object-oriented features not found in C. C++ was also invented at Bell Laboratories.
C#	Pronounced “C sharp.” It is a language invented by Microsoft for developing applications based on the Microsoft .NET platform.
Java	Java is an object-oriented language invented at Sun Microsystems and is now owned by Oracle. It may be used to develop stand-alone applications that operate on a single computer, or applications that run over the Internet from a server.
JavaScript	JavaScript is a programming language that can be used in a website to perform simple operations. Despite its name, JavaScript is not related to Java.
Perl	A general-purpose programming language that is widely used on Internet servers.
PHP	A programming language used primarily for developing web server applications and dynamic web pages.
	Python is an object-oriented programming language

Python	that is used in both business and academia. Many popular websites have features that are developed in Python.
Ruby	Ruby is a simple but powerful object-oriented programming language. It can be used for a variety of purposes, from small utility programs to large web applications.
Visual Basic	Visual Basic is a Microsoft programming language and software development environment that allows programmers to quickly create Windows-based applications.

A History of Java

In 1991, a team was formed at Sun Microsystems (a company now owned by Oracle) to speculate about the important technological trends that might emerge in the near future. The team, which was named the Green Team, concluded that computers would merge with consumer appliances. Their first project was to develop a handheld device named *7 (pronounced “star seven”) that could be used to control a variety of home entertainment devices. In order for the unit to work, it had to use a programming language that could be processed by all the devices it controlled. This presented a problem because different brands of consumer devices use different processors, each with its own machine language.

Because no such universal language existed, James Gosling, the team’s lead engineer, created one. Programs written in this language, which was originally named Oak, were not translated into the machine language of a specific processor, but were translated into an intermediate language known as *byte code*. Another program would then translate the byte code into machine language that could be executed by the processor in a specific consumer device.

Unfortunately, the technology developed by the Green Team was ahead of its time. No customers could be found, mostly because the computer-controlled

consumer appliance industry was just beginning. But rather than abandoning their hard work and moving on to other projects, the team saw another opportunity: the Internet. The Internet is a perfect environment for a universal programming language such as Oak. It consists of numerous different computer platforms connected together in a single network.

To demonstrate the effectiveness of their language, which was renamed Java, the team used it to develop a Web browser. The browser, named HotJava, was able to download and run small Java programs known as applets. This gave the browser the capability to display animation and interact with the user. HotJava was demonstrated at the 1995 SunWorld conference before a wowed audience. Later, the announcement was made that Netscape would incorporate Java technology into its Navigator browser. Other companies rapidly followed, increasing the acceptance and influence of the Java language. Today, Java is very popular for developing web applications, mobile apps, and desktop applications.

1.5 What Is a Program Made of?

Concept:

There are certain elements that are common to all programming languages.

Language Elements

All programming languages have some attributes in common. [Table 1-2](#) lists the common elements you will find in almost every language.

Table 1-2 The common elements of a programming language

Language Element	Description
Key Words	These are words that have a special meaning in the programming language. They may be used for their intended purpose only. Key words are also known as <i>reserved words</i> .
Operators	Operators are symbols or words that perform operations on one or more operands. An <i>operand</i> is usually an item of data, such as a number.
Punctuation	Most programming languages require the use of punctuation characters. These characters serve specific purposes, such as marking the beginning or ending of a statement, or separating items in a list.

	Unlike key words, which are part of the programming language, these are words or names
Programmer-defined Names	that are defined by the programmer. They are used to identify storage locations in memory and parts of the program that are created by the programmer. Programmer-defined names are often called <i>identifiers</i> .
Syntax	These are rules that must be followed when writing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

Let's look at an example Java program and identify an instance of each of these elements. [Code Listing 1-1](#) shows the code listing with each line numbered.



Note:

The line numbers are not part of the program. They are included to help point out specific parts of the program.

Code Listing 1-1 Payroll.java

```
1 public class Payroll
2 {
3     public static void main(String[] args)
4     {
5         int hours = 40;
6         double grossPay, payRate = 25.0;
7
8         grossPay = hours * payRate;
9         System.out.println("Your gross pay is $" + grossPay);
10    }
11 }
```

Key Words (Reserved Words)

Two of Java's key words appear in line 1: `public` and `class`. In line 3, the words `public`, `static`, and `void` are all key words. The words `int` in line 5 and `double` in line 6 are also key words. These words, which are always written in lowercase, each have a special meaning in Java, and can only be used for their intended purpose. As you will see, the programmer is allowed to make up his or her own names for certain things in a program. Key words, however, are reserved, and cannot be used for anything other than their designated purpose. Part of learning a programming language is learning the commonly used key words, what they mean, and how to use them.

[Table 1-3](#) shows a list of the Java key words.

Table 1-3 The Java key words

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

Programmer-Defined Names

The words `hours`, `payRate`, and `grossPay` that appear in the program in lines 5, 6, 8, and 9 are programmer-defined names. They are not part of the Java language, but are names made up by the programmer. In this particular

program, these are the names of variables. As you will learn later in this chapter, variables are the names of memory locations that may hold data.

Operators

In line 8, the following line appears:

```
grossPay = hours * payRate;
```

The = and * symbols are both operators. They perform operations on items of data, known as operands. The * operator multiplies its two operands, which in this example are the variables hours and payRate. The = symbol is called the assignment operator. It takes the value of the expression that appears at its right and stores it in the variable whose name appears at its left. In this example, the = operator stores in the grossPay variable the result of the hours variable multiplied by the payRate variable. In other words, the statement says, “the grossPay variable is assigned the value of hours times payRate.”

Punctuation

Notice that lines 5, 6, 8, and 9 end with a semicolon. A semicolon in Java is similar to a period in English: It marks the end of a complete sentence (or *statement*, as it is called in programming jargon). Semicolons do not appear at the end of every line in a Java program, however. There are rules that govern where semicolons are required and where they are not. Part of learning Java is learning where to place semicolons and other punctuation symbols.

Lines and Statements

Often, the contents of a program are thought of in terms of lines and statements. A *line* is just that—a single line as it appears in the body of a program. [Code Listing 1-1](#) is shown with each of its lines numbered. Most of

the lines contain something meaningful; however, line 7 is empty. Blank lines are only used to make a program more readable.

A statement is a complete instruction that causes the computer to perform some action. Here is the statement that appears in line 9 of [Code Listing 1-1](#):

```
System.out.println("Your gross pay is $" + grossPay);
```

This statement causes the computer to display a message on the screen. Statements can be a combination of key words, operators, and programmer-defined names. Statements often occupy only one line in a program, but sometimes they are spread out over more than one line.

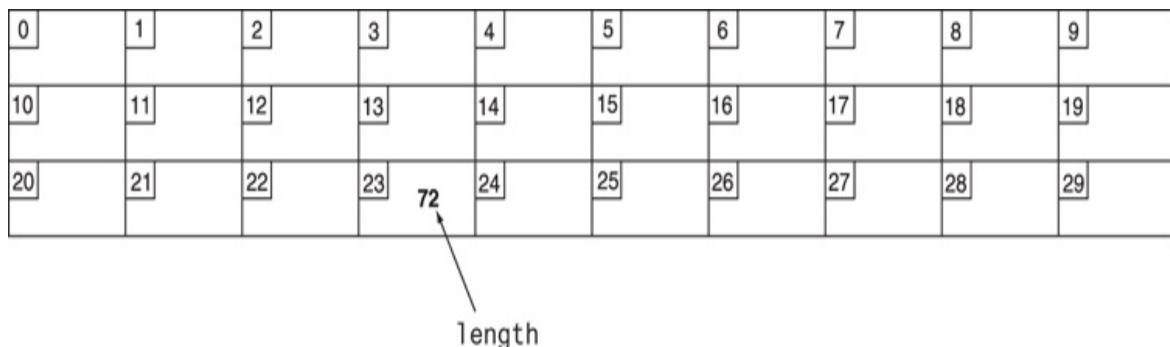
Variables

The most fundamental way that a Java program stores an item of data in memory is with a variable. A *variable* is a named storage location in the computer's memory. The data stored in a variable may change while the program is running (hence the name *variable*). Notice that in Code Listing 1-1 the programmer-defined names `hours`, `payRate`, and `grossPay` appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user has worked. The `payRate` variable stores the user's hourly pay rate. The `grossPay` variable holds the result of `hours` multiplied by `payRate`, which is the user's gross pay.

Variables are symbolic names made up by the programmer that represent locations in the computer's RAM. When data is stored in a variable, it is actually stored in RAM. Assume that a program has a variable named `length`. [Figure 1-4](#) illustrates the way the variable name represents a memory location.

**Figure 1-4 A variable name
represents a location in**

memory



In [Figure 1-4](#), the variable `length` is holding the value 72. The number 72 is actually stored in RAM at address 23, but the name `length` symbolically represents this storage location. If it helps, you can think of a variable as a box that holds data. In [Figure 1-4](#), the number 72 is stored in the box named `length`. Only one item may be stored in the box at any given time. If the program stores another value in the box, it will take the place of the number 72.

The Compiler and the Java Virtual Machine

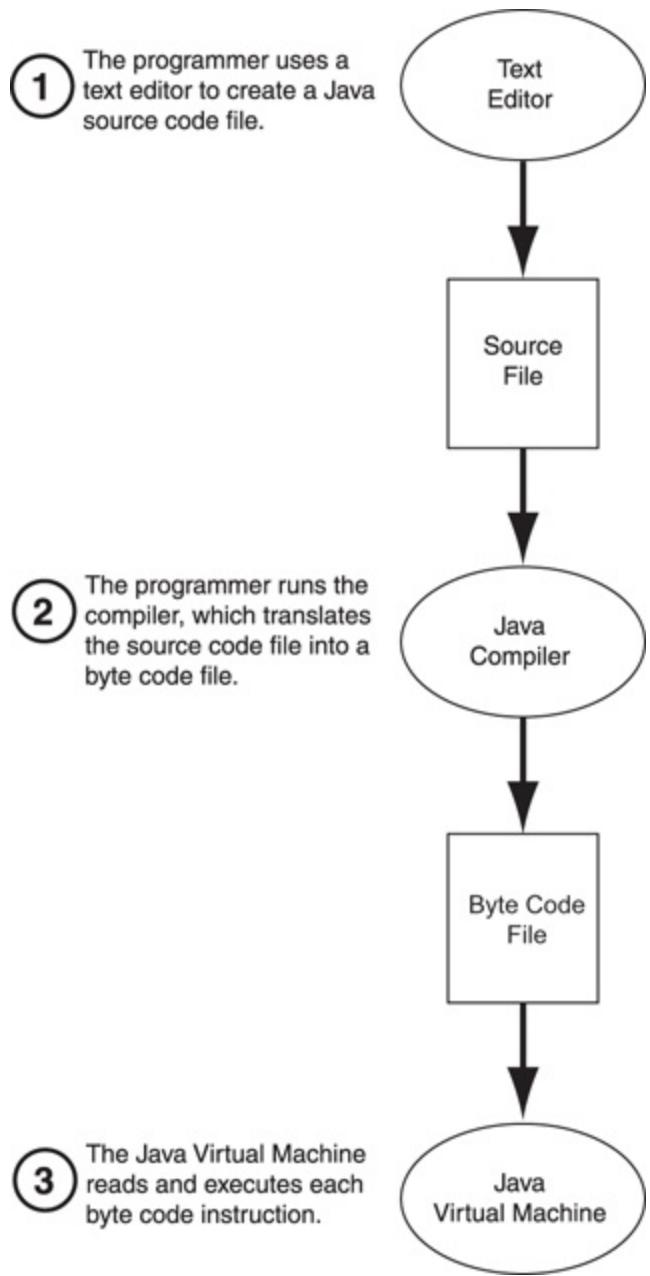
When a Java program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The Java programming statements written by the programmer are called *source code*, and the file they are saved in is called a *source file*. Java source files end with the `.java` extension.

After the programmer saves the source code to a source file, he or she runs the Java compiler. A compiler is a program that translates source code into an executable form. During the translation process, the compiler uncovers any syntax errors that may be in the program. Syntax errors are mistakes that the programmer has made that violate the rules of the programming language. These errors must be corrected before the compiler can translate the source

code. Once the program is free of syntax errors, the compiler creates another file that holds the translated instructions.

Most programming language compilers translate source code directly into files that contain machine language instructions. These files are called executable files because they may be executed directly by the computer's CPU. The Java compiler, however, translates a Java source file into a file that contains byte code instructions. Byte code instructions are not machine language, and therefore cannot be directly executed by the CPU. Instead, they are executed by the Java Virtual Machine. The Java Virtual Machine (JVM) is a program that reads Java byte code instructions and executes them as they are read. For this reason, the JVM is often called an interpreter, and Java is often referred to as an interpreted language. [Figure 1-5](#) illustrates the process of writing a Java program, compiling it to byte code, and running it.

Figure 1-5 Program development process



[Figure 1-5 Full Alternative Text](#)

Although Java byte code is not machine language for a CPU, it can be considered as machine language for the JVM. You can think of the JVM as a program that simulates a computer whose machine language is Java byte code.

Portability

The term *portable* means that a program may be written on one type of computer then run on a wide variety of computers, with little or no required modification. Because Java byte code is the same on all computers, compiled Java programs are highly portable. In fact, a compiled Java program may be run on any computer that has a Java Virtual Machine. [Figure 1-6](#) illustrates the concept of a compiled Java program running on Windows, Linux, Mac, and UNIX computers.

Figure 1-6 Java byte code may be run on any computer with a JVM

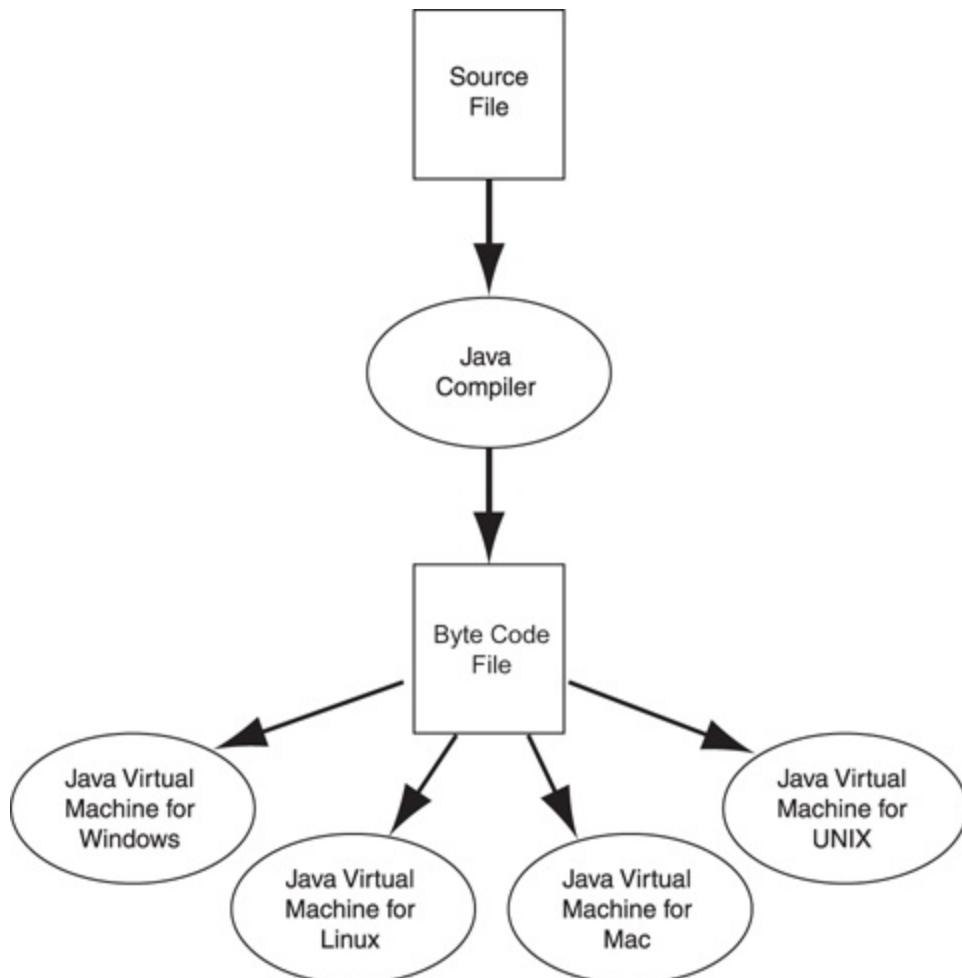


Figure 1-6 Full Alternative Text

With most other programming languages, portability is achieved by the creation of a compiler for each type of computer that the language is to run on. For example, in order for the C++ language to be supported by Windows, Linux, and Mac computers, a separate C++ compiler must be created for each of those environments. Compilers are very complex programs and more difficult to develop than interpreters. For this reason, a JVM has been developed for many types of computers.

Java Software Editions

The software used to create Java programs is referred to as the *JDK* (Java Development Kit) or the *SDK* (Software Development Kit). These are the different editions of the JDK available from Oracle:

- Java SE—The Java Standard Edition provides all the essential software tools necessary for writing Java applications.
- Java EE—The Java Enterprise Edition provides tools for creating large business applications that employ servers and provide services over the Web.
- Java ME—The Java Micro Edition provides a small, highly optimized runtime environment for consumer products such as cell phones, television DVRs, and appliances.

These editions of Java may be downloaded from Oracle at:

<http://java.oracle.com>



Note:

You can follow the instructions in Appendix D, which can be downloaded

from the book's companion website, to install the JDK on your system. You can access the book's companion website by going to www.pearsonhighered.com/gaddis.

Compiling and Running a Java Program

Compiling a Java program is a simple process. To use the JDK, go to your operating system's command prompt.

Tip:

In Windows 8 and Windows 10, press WIN+X on the keyboard, and then select *Command Prompt* from the menu. A command prompt window should appear.



VideoNote Compiling and Running a Java Program

At the operating system command prompt, make sure you are in the same directory or folder where the Java program that you want to compile is located. Then, use the `javac` command, in the following form:

```
javac Filename
```

Filename is the name of a file that contains the Java source code. As mentioned earlier, this file has the `.java` extension. For example, if you want to compile the `Payroll.java` file, you would execute the following command:

```
javac Payroll.java
```

This command runs the compiler. If the file contains any syntax errors, you will see one or more error messages, and the compiler will not translate the file to byte code. When this happens, you must open the source file in a text

editor and fix the error. Then you can run the compiler again. If the file has no syntax errors, the compiler will translate it to byte code. Byte code is stored in a file with the *.class* extension, so the byte code for the *Payroll.java* file will be stored in *Payroll.class*, which will be in the same directory or folder as the source file.

To run the Java program, you use the `java` command in the following form:

```
java ClassFilename
```

ClassFilename is the name of the *.class* file that you wish to execute. However, you do not type the *.class* extension. For example, to run the program that is stored in the *Payroll.class* file, you would enter the following command:

```
java Payroll
```

This command runs the Java interpreter (the JVM) and executes the program.

Integrated Development Environments



VideoNote Using an IDE

In addition to the command prompt programs, there are also several Java integrated development environments (IDEs). These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. A program is compiled and executed with a single click of a button, or by selecting a single item from a menu. [Figure 1-7](#) shows a screen from the NetBeans IDE.

Figure 1-7 The NetBeans IDE

The screenshot shows the NetBeans IDE 8.1 interface. The title bar reads "Payroll - NetBeans IDE 8.1". The menu bar includes File, Edit, View, Navigate, Source, Refactor, Run, Debug, Profile, Team, Tools, Window, Help. A search bar at the top right says "Search (Ctrl+I)". Below the menu is a toolbar with various icons. The left sidebar shows a project tree with "Payroll" selected, containing "Source Packages" with "payroll" and "Payroll.java". The main editor window displays the following Java code:

```
public class Payroll
{
    public static void main (String[] args)
    {
        int hours = 40;
        double grossPay, payRate = 25.0;

        grossPay = hours * payRate;
        System.out.printIn("Your gross pay is $" + grossPay);
    }
}
```

The Navigator panel on the left shows "Members" and "Payroll". The bottom status bar shows "12:1" and "INS".

[Figure 1-7 Full Alternative Text](#)

Checkpoint

1. 1.8 Describe the difference between a key word and a programmer-defined symbol.
2. 1.9 Describe the difference between operators and punctuation symbols.
3. 1.10 Describe the difference between a program line and a statement.

4. 1.11 Why are variables called “variable”?
5. 1.12 What happens to a variable’s current contents when a new value is stored there?
6. 1.13 What is a compiler?
7. 1.14 What is a syntax error?
8. 1.15 What is byte code?
9. 1.16 What is the JVM?

1.6 The Programming Process

Concept:

The programming process consists of several steps which include design, creation, testing, and debugging activities.

Now that you have been introduced to what a program is, it's time to consider the process of creating a program. Quite often when inexperienced students are given programming assignments, they have trouble getting started because they don't know what to do first. If you find yourself in this dilemma, the following steps may help:

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.
4. Check the model for logical errors.
5. Enter the code and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

These steps emphasize the importance of planning. Just as there are good ways and bad ways to paint a house, there are good ways and bad ways to create a program. A good program always begins with planning. With the pay-calculating algorithm that was presented earlier in this chapter serving as our example, let's look at each of the steps in more detail.

1. Clearly define what the program is to do.

This step commonly requires you to identify the purpose of the program, the data that is to be input, the processing that is to take place, and the desired output. Let's examine each of these requirements for the pay-calculating algorithm.

Purpose To calculate the user's gross pay.

Input Number of hours worked, hourly pay rate.

Process Multiply number of hours worked by hourly pay rate. The result is the user's gross pay.

Output Display a message indicating the user's gross pay.

2. Visualize the program running on the computer.

Before you create a program on the computer, you should first create it in your mind. Try to imagine what the computer screen will look like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program. For instance, [Figure 1-8](#) shows the screen we might want produced by a program that implements the pay-calculating algorithm.

Figure 1-8 Screen produced by the pay-calculating algorithm

```
How many hours did you work? 10  
How much do you get paid per hour? 15  
Your gross pay is $150.0
```

[Figure 1-8 Full Alternative Text](#)

In this step, you must put yourself in the shoes of the user. What messages should the program display? What questions should it ask? By addressing these concerns, you can determine most of the program's output.

3. Use design tools to create a model of the program.

While planning a program, the programmer uses one or more design tools to create a model of the program. For example, *pseudocode* is a cross between human language and a programming language and is especially helpful when designing an algorithm. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm in a language that's "almost" a programming language, but still very similar to natural language. For example, here is pseudocode that describes the pay-calculating algorithm:

- *Get payroll data.*
- *Calculate gross pay.*
- *Display gross pay.*

Although this pseudocode gives a broad view of the program, it does not reveal all the program's details. A more detailed version of the pseudocode follows:

- *Display “How many hours did you work?”*
- *Input hours.*
- *Display “How much do you get paid per hour?”*
- *Input rate.*
- *Store the value of hours times rate in the pay variable.*
- *Display the value in the pay variable.*

Notice that the pseudocode uses statements that look more like commands than the English statements that describe the algorithm in [Section 1.4](#). The pseudocode even names variables and describes mathematical operations.

4. Check the model for logical errors.

Logical errors are mistakes that cause the program to produce erroneous results. Once a model of the program is assembled, it should be checked for these errors. For example, if pseudocode is used, the programmer should trace through it, checking the logic of each step. If an error is found, the model can be corrected before the next step is attempted.

5. Enter the code and compile it.

Once a model of the program has been created, checked, and corrected, the programmer is ready to write source code on the computer. The programmer saves the source code to a source file and begins the process of compiling it.

During this step, the compiler will find any syntax errors that may exist in the program.

6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.

If the compiler reports any errors, they must be corrected. Steps 5 and 6 must be repeated until the program is free of compile-time errors.

7. Run the program with test data for input.

Once an executable file is generated, the program is ready to be tested for runtime errors. A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for runtime errors requires that the program be executed with sample data or sample input. The sample data should be such that the correct output can be predicted. If the program does not produce the correct output, a logical error is present in the program.

8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.

When runtime errors are found in a program, they must be corrected. You must identify the step where the error occurred and determine the cause. If an error is a result of incorrect logic (such as an improperly stated math formula), you must correct the statement or statements involved in the logic. If an error is due to an incomplete understanding of the program requirements, then you must restate the program purpose and modify the program model and source code. The program must then be saved, recompiled, and retested. This means Steps 5 through 8 must be repeated until the program reliably produces satisfactory results.

9. Validate the results of the program.

When you believe you have corrected all the runtime errors, enter test data and determine if the program solves the original problem.



Checkpoint

1. 1.17 What four items should you identify when defining what a program is to do?
2. 1.18 What does it mean to “visualize a program running”? What is the value of such an activity?
3. 1.19 What is pseudocode?
4. 1.20 Describe what a compiler does with a program’s source code.
5. 1.21 What is a runtime error?
6. 1.22 Is a syntax error (such as misspelling a key word) found by the compiler or when the program is running?
7. 1.23 What is the purpose of testing a program with sample data or input?

1.7 Object-Oriented Programming

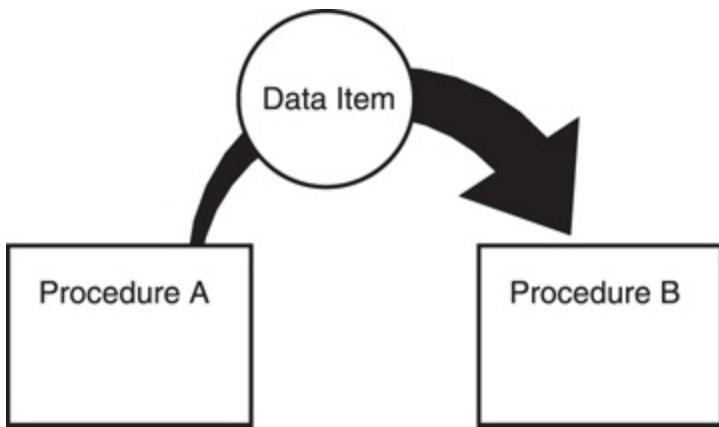
Concept:

Java is an object-oriented programming (OOP) language. OOP is a method of software development that has its own practices, concepts, and vocabulary.

There are primarily two methods of programming in use today: procedural, and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. A *procedure* is a set of programming statements that, together, perform a specific task. The statements might gather input from the user, manipulate data stored in the computer's memory, and perform calculations or any other operation necessary to complete its task.

Procedures typically operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another, as shown in [Figure 1-9](#).

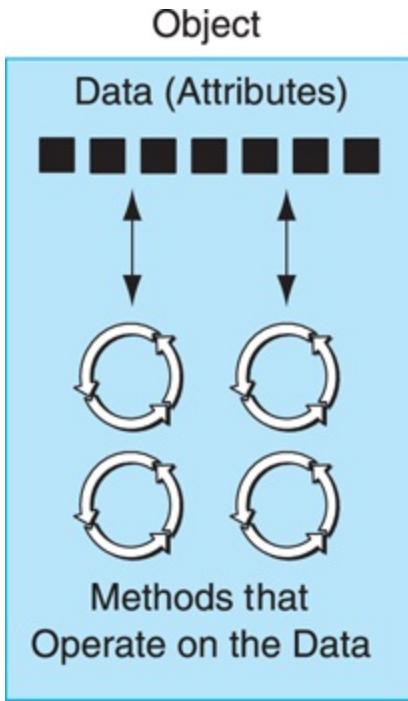
Figure 1-9 Data is passed among procedures



As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data often leads to problems, however. For example, the data is stored in a particular format, which consists of variables and more complex structures that are created from variables. The procedures that operate on the data must be designed with that format in mind. But, what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in a redesigned data format. When the structure of the data changes, the code that operates on the data must also be changed to accept the new format. This results in additional work for programmers, and a greater opportunity for bugs to appear in the code.

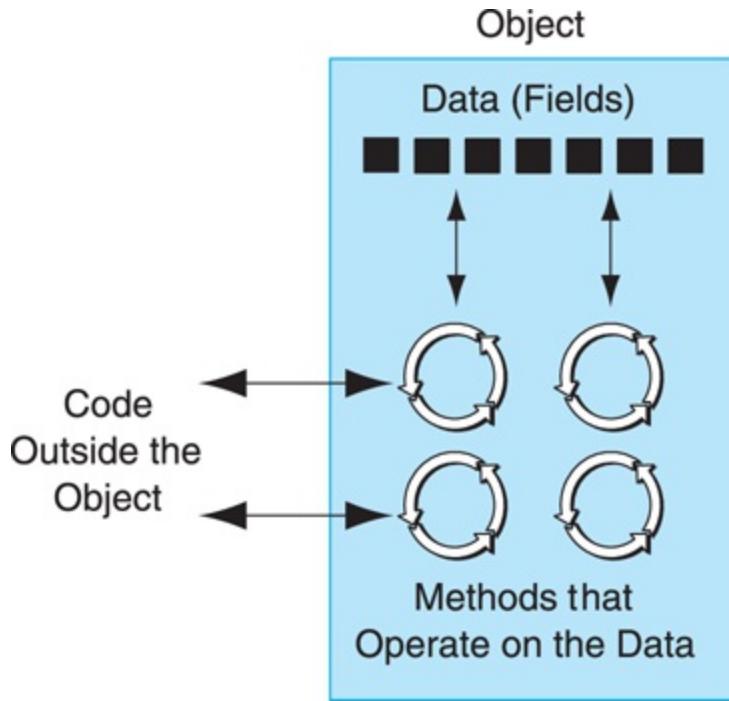
This has helped influence the shift from procedural programming to OOP. Whereas procedural programming is centered on creating procedures, object-oriented programming is centered on creating objects. An object is a software entity that contains data and procedures. The data contained in an object is known as the object's *attributes*. The procedures, or behaviors, that an object performs are known as the object's *methods*. The object is, conceptually, a self-contained unit consisting of data (attributes) and procedures (methods). This is illustrated in [Figure 1-10](#).

Figure 1-10 An object contains data and procedures



OOP addresses the problem of code/data separation through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code that is outside the object. Only the object's methods may then directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access the methods that operate on the data. As shown in [Figure 1-11](#), the object's methods provide programming statements outside the object indirect access to the object's data.

Figure 1-11 Code outside the object interacts with the object's methods



When an object's internal data is hidden from outside code and access to that data is restricted to the object's methods, the data is protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or internal structure of the object's data. The code only needs to interact with the object's methods. When a programmer changes the structure of an object's internal data, he or she also modifies the object's methods so they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

Component Reusability

In addition to solving the problems of code and data separation, the use of OOP has also been encouraged by the trend of *component reusability*. A component is a software object that performs a specific, well-defined operation or that provides a particular service. The component is not a stand-alone program, but can be used by programs that need the component's service. For example, Sharon is a programmer who has developed a component for rendering three-dimensional (3D) images. She is a math whiz and knows a lot about computer graphics, so her component is coded to perform all the necessary 3D mathematical operations and handle the

computer's video hardware. Tom, who is writing a program for an architectural firm, needs his application to display 3D images of buildings. Because he is working under a tight deadline and does not possess a great deal of knowledge about computer graphics, he can use Sharon's component to perform the 3D rendering (for a small fee, of course!).

Component reusability and OOP technology set the stage for large-scale computer applications to become systems of unique collaborating entities (components).

An Everyday Example of an Object

Think of your alarm clock as an object. It has the following attributes:

- The current second (a value in the range of 0–59)
- The current minute (a value in the range of 0–59)
- The current hour (a value in the range of 1–12)
- The time the alarm is set for (a valid hour and minute)
- Whether the alarm is on or off (“on” or “off”)

As you can see, the attributes are merely data values that define the alarm clock's state. You, the user of the alarm clock object, cannot directly manipulate these attributes because they are *private*. To change an attribute's value, you must use one of the object's methods. Here are some of the alarm clock object's methods:

- Set time
- Set alarm time
- Turn alarm on
- Turn alarm off

Each method manipulates one or more of the attributes. For example, the “set time” method allows you to set the alarm clock’s time. You activate the method by pressing a set of buttons on top of the clock. By using another set of buttons, you can activate the “set alarm time” method. In addition, another button allows you to execute the “turn alarm on” and “turn alarm off” methods. Notice that all of these methods can be activated by you, who are outside of the alarm clock. Methods that can be accessed by entities outside the object are known as *public methods*.

The alarm clock also has *private methods*, which are part of the object’s private, internal workings. External entities (such as you, the user of the alarm clock) do not have direct access to the alarm clock’s private methods. The object is designed to execute these methods automatically and hide the details from you. Here are the alarm clock object’s private methods:

- Increment the current second
- Increment the current minute
- Increment the current hour
- Sound alarm

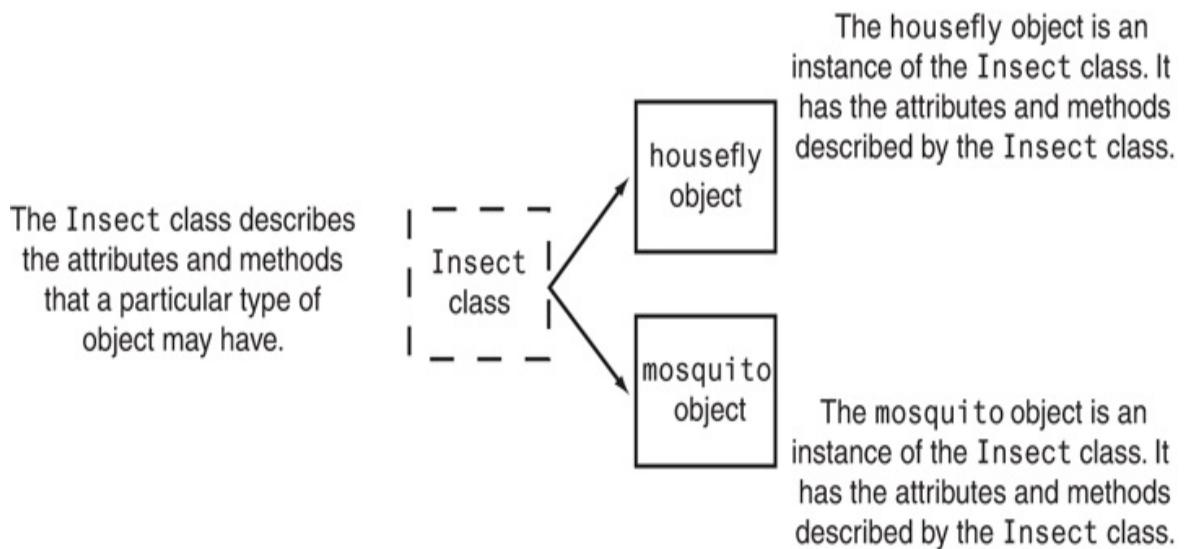
Every second, the “increment the current second” method executes. This changes the value of the current second attribute. If the current second attribute is set to 59 when this method executes, the method is programmed to reset the current second to 0, then cause the “increment current minute” method to execute. This method adds 1 to the current minute, unless it is set to 59. In that case, it resets the current minute to 0 and causes the “increment current hour” method to execute. (It might also be noted that the “increment current minute” method compares the new time to the alarm time. If the two times match and the alarm is turned on, the “sound alarm” method is executed.)

Classes and Objects

Now let us discuss how objects are created in software. Before an object can be created, it must be designed by a programmer. The programmer determines the attributes and methods that are necessary, then creates a class. A *class* is a collection of programming statements that specify the attributes and methods that a particular type of object may have. Think of a class as a “blueprint” that objects may be created from. So, a class is not an object, but a description of an object. When the program is running, it can use the class to create, in memory, as many objects as needed. Each object that is created from a class is called an *instance* of the class.

For example, Jessica is an entomologist (someone who studies insects), and she also enjoys writing computer programs. She designs a program to catalog different types of insects. In the program, she creates a class named `Insect`, which specifies attributes and methods for holding and manipulating data common to all types of insects. The `Insect` class is not an object, but a specification that objects may be created from. Next, she writes programming statements that create a `housefly` object, which is an instance of the `Insect` class. The `housefly` object is an entity that occupies computer memory and stores data about a `housefly`. It has the attributes and methods specified by the `Insect` class. Then she writes programming statements that create a `mosquito` object. The `mosquito` object is also an instance of the `Insect` class. It has its own area in memory and stores data about a mosquito. Although the `housefly` and `mosquito` objects are two separate entities in the computer’s memory, they were both created from the `Insect` class. This means that each of the objects have the attributes and methods described by the `Insect` class. This is illustrated in [Figure 1-12](#).

Figure 1-12 The housefly and mosquito objects are instances of the Insect class

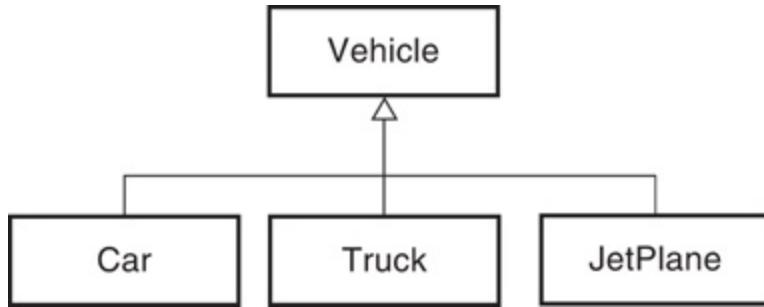


[Figure 1-12 Full Alternative Text](#)

Inheritance

Sometimes a class is based on another class. This means that one class is a specialized case of the other. For example, consider a program that uses classes representing cars, trucks, and jet planes. Although those three types of objects in the real world are very different, they have some common characteristics: They are all modes of transportation, and they all can carry some number of passengers. So, each of the three classes could be based on a Vehicle class that has the attributes and behaviors common to all of the classes. This is illustrated in [Figure 1-13](#).

Figure 1-13 An example of inheritance



In OOP terminology, the `Vehicle` class is the *superclass*. The `Car`, `Truck`, and `JetPlane` classes are *subclasses*. Although the `Vehicle` class is very general in nature, the `Car`, `Truck`, and `JetPlane` classes are specialized. All of the attributes and behaviors of the `Vehicle` class are inherited by the `Car`, `Truck`, and `JetPlane` classes. The relationship between the classes implies that a `Car` is a `Vehicle`, a `Truck` is a `Vehicle`, and a `JetPlane` is a `Vehicle`.

In addition to inheriting the attributes and methods of the superclass, subclasses add their own. For example, the `Car` class might have attributes and methods that set and indicate whether it is a sedan or coupe, and the type of engine it has. The `Truck` class might have attributes and methods that set and indicate the maximum amount of weight it can carry, and the number of miles it can travel between refueling. The `JetPlane` class might have attributes and methods that set and indicate the plane's altitude and heading. These added capabilities make the subclasses more specialized than the superclass.

Software Engineering

The field of software engineering encompasses the entire process of crafting computer software. It includes designing, writing, testing, debugging, documenting, modifying, and maintaining complex software development projects. Like traditional engineers, software engineers use a number of tools in their craft. Here are a few examples:

- Program specifications
- Diagrams of screen output

- Diagrams representing classes, objects, and the flow of data
- Pseudocode
- Examples of expected input and desired output
- Special software designed for testing programs

Most commercial software applications are large and complex. Usually a team of programmers, not a single individual, develops them. It is important that the program requirements be thoroughly analyzed and divided into subtasks that are handled by individual teams, or individuals within a team.



Checkpoint

1. 1.24 In procedural programming, what two parts of a program are typically separated?
2. 1.25 What are an object's attributes?
3. 1.26 What are an object's methods?
4. 1.27 What is encapsulation?
5. 1.28 What is data hiding?

Review Questions and Exercises

Multiple Choice

1. This part of the computer fetches instructions, carries out the operations commanded by the instructions, and produces some outcome or resultant information.
 1. memory
 2. CPU
 3. secondary storage
 4. input device
2. A byte is made up of eight .
 1. CPUs
 2. addresses
 3. variables
 4. bits
3. Each byte is assigned a unique .
 1. address
 2. CPU
 3. bit
 4. variable

4. This type of memory can hold data for long periods of time—even when there is no power to the computer.
 1. RAM
 2. primary storage
 3. secondary storage
 4. CPU storage
5. If you were to look at a machine language program, you would see
 - .
1. Java source code
2. a stream of binary numbers
3. English words
4. circuits
6. These are words that have a special meaning in the programming language.
 1. punctuation
 2. programmer-defined names
 3. key words
 4. operators
7. These are symbols or words that perform operations on one or more operands.
 1. punctuation
 2. programmer-defined names

3. key words
 4. operators
8. These characters serve specific purposes, such as marking the beginning or ending of a statement, or separating items in a list.
 1. punctuation
 2. programmer-defined names
 3. key words
 4. operators
9. These are words or names that are used to identify storage locations in memory and parts of the program that are created by the programmer.
 1. punctuation
 2. programmer-defined names
 3. key words
 4. operators
10. These are the rules that must be followed when writing a program.
 1. syntax
 2. punctuation
 3. key words
 4. operators
11. This is a named storage location in the computer's memory.
 1. class

2. key word

3. variable

4. operator

12. The Java compiler generates _____.

1. machine code

2. byte code

3. source code

4. HTML

13. JVM stands for _____.

1. Java Variable Machine

2. Java Variable Method

3. Java Virtual Method

4. Java Virtual Machine

Find the Error

1. The following pseudocode algorithm has an error. The program is supposed to ask the user for the length and width of a rectangular room, then display the room's area. The program must multiply the width by the length in order to determine the area. Find the error.

- $area = width \times length$

- *Display “What is the room’s width?”*

- *Input width.*

- *Display* “*What is the room’s length?*”
- *Input length.*
- *Display area.*

Algorithm Workbench

Write pseudocode algorithms for the programs described as follows:

1. Available Credit

A program that calculates a customer’s available credit should ask the user for the following:

- The customer’s maximum amount of credit
- The amount of credit used by the customer

Once these items have been entered, the program should calculate and display the customer’s available credit. You can calculate available credit by subtracting the amount of credit used from the maximum amount of credit.

2. Sales Tax

A program that calculates the total of a retail sale should ask the user for the following:

- The retail price of the item being purchased
- The sales tax rate

Once these items have been entered, the program should calculate and display the following:

- The sales tax for the purchase

- The total of the sale

3. Account Balance

A program that calculates the current balance in a savings account must ask the user for the following:

- The starting balance
- The total dollar amount of deposits made
- The total dollar amount of withdrawals made
- The monthly interest rate

Once the program calculates the current balance, it should be displayed on the screen.

Predict the Result

The following are programs expressed as English statements. What would each display on the screen if they were actual programs?

1. The variable x starts with the value 0.
 - The variable y starts with the value 5.
 - Add 1 to x .
 - Add 1 to y .
 - Add x and y , and store the result in y .
 - Display the value in y on the screen.
2. The variable a starts with the value 10.
 - The variable b starts with the value 2.

- The variable c starts with the value 4.
- Store the value of a times b in a.
- Store the value of b times c in c.
- Add a and c, and store the result in b.
- Display the value in b on the screen.

Short Answer

1. Both main memory and secondary storage are types of memory.
Describe the difference between the two.
2. What type of memory is usually volatile?
3. What is the difference between operating system software and application software?
4. Indicate all the categories that the following operating systems belong to:

System This system allows multiple users to run multiple programs
 A simultaneously.

System Only one user may access the system at a time, but multiple
 B programs can be run simultaneously.

System Only one user may access the system at a time, and only one
 C program can be run on the system at a time.

5. Why must programs written in a high-level language be translated into machine language before they can be run?
6. Why is it easier to write a program in a high-level language than in machine language?

7. What is a source file?
8. What is the difference between a syntax error and a logical error?
9. What is an algorithm?
10. What is a compiler?
11. What must a computer have in order for it to execute Java programs?
12. What is the difference between machine language code and byte code?
13. Why does byte code make Java a portable language?
14. Is encapsulation a characteristic of procedural or object-oriented programming?
15. Why should an object hide its data?
16. What part of an object forms an interface through which outside code may access the object's data?
17. What is component reusability?
18. What is a class?
19. How is a class different from an object?
20. What object-oriented programming characteristic allows you to create a class that is a specialized version of another class?
21. What type of program do you use to write Java source code?
22. Will the Java compiler translate a source file that contains syntax errors?
23. What does the Java compiler translate Java source code to?
24. Assuming you have installed the Java JDK, what command would you type at the operating system command prompt to compile the program

LabAssignment.java?

25. Assuming there are no syntax errors in the *LabAssignment.java* program when it is compiled, answer the following questions.

1. What file will be produced?
2. What will the file contain?
3. What command would you type at the operating system command prompt to run the program?

Programming Challenge

1. 1. Your First Java Program



VideoNote Your First Java Program

This assignment will help you get acquainted with your Java development software. Here is the Java program you will enter:

```
// This is my first Java program.  
public class MyFirstProgram  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```

2. If You Are Using the JDK at the Command Prompt:

1. Use a text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters, and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
2. After saving the program, go to your operating system's command prompt and change your current directory or folder to the one that contains the Java program you just created. Then, use the following command to compile the program:

```
javac MyFirstProgram.java
```

If you typed the contents of the file exactly as shown, you shouldn't have any syntax errors. If you see error messages, open the file in the editor and compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again.

If you see no error messages, the file was successfully compiled.

3. Next, enter the following command to run the program:

```
java MyFirstProgram
```

Be sure to use the capitalization of `MyFirstProgram` exactly as it is shown here. You should see the message “Hello World!” displayed on the screen.

3. If You Are Using an IDE:

Because there are many Java IDEs, we cannot include specific instructions for all of these. The following are general steps that should apply to most of them. You will need to consult your IDE’s documentation for specific instructions.

1. Start your Java IDE and perform any necessary setup operations, such as starting a new project and creating a new Java source file.
2. Use the IDE’s text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters, and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
3. After saving the program, use your IDE’s command to compile the program. If you typed the contents of the file exactly as shown, you shouldn’t have any syntax errors. If you see error messages, compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again. If you see no error messages, the file was successfully compiled.

Use your IDE’s command to run the program. You should see the message “Hello World!” displayed.

Chapter 2 Java Fundamentals

Topics

1. [2.1 The Parts of a Java Program](#)
2. [2.2 The `System.out.print` and `System.out.println` Methods, and the Java API](#)
3. [2.3 Variables and Literals](#)
4. [2.4 Primitive Data Types](#)
5. [2.5 Arithmetic Operators](#)
6. [2.6 Combined Assignment Operators](#)
7. [2.7 Conversion between Primitive Data Types](#)
8. [2.8 Creating Named Constants with `final`](#)
9. [2.9 The `String` Class](#)
10. [2.10 Scope](#)
11. [2.11 Comments](#)
12. [2.12 Programming Style](#)
13. [2.13 Reading Keyboard Input](#)
14. [2.14 Dialog Boxes](#)
15. [2.15 Displaying Formatted Output with `System.out.printf` and `String.format`](#)

16. [2.16 Common Errors to Avoid](#)

2.1 The Parts of a Java Program

Concept:

A Java program has parts that serve specific purposes.

Java programs are made up of different parts. Your first step in learning Java is to learn what the parts are. We will begin by looking at a simple example, shown in [Code Listing 2-1](#).

Code Listing 2-1 (`Simple.java`)

```
1 // This is a simple Java program.  
2  
3 public class Simple  
4 {  
5     public static void main(String[] args)  
6     {  
7         System.out.println("Programming is great fun!");  
8     }  
9 }
```



Remember, the line numbers shown in the program listings are not part of the program. The numbers are shown so we can refer to specific lines in the programs.

As mentioned in [Chapter 1](#), the names of Java source code files end with `.java`. The program shown in [Code Listing 2-1](#) is named `Simple.java`. Using the Java compiler, this program may be compiled with the following

command:

```
javac Simple.java
```

The compiler will create another file named *Simple.class*, which contains the translated Java byte code. This file can be executed with the following command:

```
java Simple
```

Tip:

Remember, you do not type the *.class* extension when using the `java` command.

The output of the program is as follows. This is what appears on the screen when the program runs.

Programming is great fun!

Let's examine the program line by line. Here's the statement in line 1:

```
// This is a simple Java program.
```

Other than the two slash marks that begin this line, it looks pretty much like an ordinary sentence. The `//` marks the beginning of a comment. The compiler ignores everything from the double slash to the end of the line. That means you can type anything you want on that line, and the compiler never complains. Although comments are not required, they are very important to programmers. Most programs are much more complicated than this example, and comments help explain what's going on.

Line 2 is blank. Programmers often insert blank lines in programs to make them easier to read. Line 3 reads:

```
public class Simple
```

This line is known as a *class header*, and it marks the beginning of a *class*

definition. One of the uses of a class is to serve as a container for an application. As you progress through this book, you will learn more and more about classes. For now, just remember that a Java program must have at least one class definition. This line of code consists of three words: `public`, `class`, and `Simple`. Let's take a closer look at each word.

- `public` is a Java key word, and it must be written in all lowercase letters. It is known as an *access specifier*, and it controls where the class may be accessed from. The `public` specifier means access to the class is unrestricted. (In other words, the class is “open to the public.”)
- `class`, which must also be written in lowercase letters, is a Java key word that indicates the beginning of a class definition.
- `Simple` is the class name. This name was made up by the programmer. The class could have been called `Pizza`, or `Dog`, or anything else the programmer wanted. Programmer-defined names may be written in lowercase letters, uppercase letters, or a mixture of both.

In a nutshell, this line of code tells the compiler that a publicly accessible class named `Simple` is being defined. Here are two more points to know about classes:

- You may create more than one class in a file, but you may have only one `public class` per Java file.
- When a Java file has a `public class`, the name of the `public class` must be the same as the name of the file (without the `.java` extension). For instance, the program in [Code Listing 2-1](#) has a `public class` named `Simple`, so it is stored in a file named `Simple.java`.



Note:

Java is a case-sensitive language. That means it regards uppercase letters as entirely different characters than their lowercase counterparts. The word `Public` is not the same as `public`, and `Class` is not the same as `class`. Some

words in a Java program must be entirely in lowercase, while other words may use a combination of lower and uppercase characters. Later in this chapter, you will see a list of all the Java key words, which must appear in lowercase.

Line 4 contains only a single character:

{

This is called a left brace, or an opening brace, and is associated with the beginning of the class definition. All of the programming statements that are part of the class are enclosed in a set of braces. If you glance at the last line in the program, line 9, you'll see the closing brace. Everything between the two braces is the *body* of the class named `Simple`. Here is the program code again; this time the body of the class definition is shaded.

```
// This is a simple Java program.  
public class Simple  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Programming is great fun!");  
    }  
}
```



Warning!

Make sure you have a closing brace for every opening brace in your program!

Line 5 reads:

```
public static void main(String[] args)
```

This line is known as a *method header*. It marks the beginning of a *method*. A method can be thought of as a group of one or more programming statements that collectively has a name. When creating a method, you must tell the compiler several things about it. That is why this line contains so many words. At this point, the only thing you should be concerned about is that the

name of the method is `main`, and the rest of the words are required for the method to be properly defined. This is shown in [Figure 2-1](#).

Figure 2-1 The `main` method header

Name of the Method
↓
`public static void main (string [] args)`

The other parts of this line are necessary
for the method to be properly defined.

[Figure 2-1 Full Alternative Text](#)

Recall from [Chapter 1](#) that a stand-alone Java program that runs on your computer is known as an application. Every Java application must have a method named `main`. The `main` method is the starting point of an application.



Note:

For the time being, all the programs you will write will consist of a class with a `main` method whose header looks exactly like the one shown in [Code Listing 2-1](#). As you progress through this book, you will learn what `public static void` and `(String[] args)` mean. For now, just assume that you are learning a “recipe” for assembling a Java program.

Line 6 has another opening brace:

{

This opening brace belongs to the `main` method. Remember that braces enclose statements, and every opening brace must have an accompanying

closing brace. If you look at line 8, you will see the closing brace that corresponds to this opening brace. Everything between these braces is the *body* of the `main` method.

Line 7 appears as follows:

```
System.out.println("Programming is great fun!");
```

To put it simply, this line displays a message on the screen. The message “Programming is great fun!” is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal*.



Note:

This is the only line in the program that causes anything to be printed on the screen. The other lines, like `public class Simple` and `public static void main(String[] args)`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

At the end of the line is a *semicolon*. Just as a period marks the end of a sentence, a semicolon marks the end of a statement in Java. Not every line of code ends with a semicolon, however. Here is a summary of instances in which you do not place a semicolon:

- Comments do not have to end with a semicolon because they are ignored by the compiler.
- Class headers and method headers do not end with a semicolon because they are terminated with a body of code inside braces.
- The brace characters, { and }, are not statements, so you do not place a semicolon after them.

It might seem that the rules for where to put a semicolon are not clear at all. For now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

As has already been pointed out, lines 8 and 9 contain the closing braces for the `main` method and the class definition:

```
    }  
}
```

Before continuing, let's review the points we just covered, including some of the more elusive rules.

- Java is a case-sensitive language. It does not regard uppercase letters as being the same character as their lowercase equivalents.
- All Java programs must be stored in a file with a name that ends with `.java`.
- Comments are ignored by the compiler.
- A `.java` file may contain many classes, but may only have one `public` class. If a `.java` file has a public class, the class must have the same name as the file. For instance, if the file `Pizza.java` contains a public class, the class's name would be `Pizza`.
- Every Java application program must have a method named `main`.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons. This does not include comments, class headers, method headers, or braces.

In the sample program, you encountered several special characters. [Table 2-1](#) summarizes how they were used.

Table 2-1 Special characters

Characters	Name	Meaning
//	Double slash	Marks the beginning of a comment
()	Opening and closing parentheses	Used in a method header
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a class or a method
" "	Quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	Semicolon	Marks the end of a complete programming statement



Checkpoint

- 2.1 The following program will not compile because the lines have been mixed up.

```
public static void main(String[] args)
{
// A crazy mixed up program
public class Columbus
{
System.out.println("In 1492 Columbus sailed the ocean blue.")
{}
```

When the lines are properly arranged, the program should display the following on the screen:

In 1492 Columbus sailed the ocean blue.

Rearrange the lines in the correct order. Test the program by entering it

on the computer, compiling it, and running it.

2. 2.2 When the program in [Checkpoint 2.1](#) is saved to a file, what should the file be named?
3. 2.3 Complete the following program skeleton so that it displays the message “Hello World” on the screen.

```
public class Hello
{
    public static void main(String[] args)
    {
        // Insert code here to complete the program
    }
}
```

4. 2.4 On paper, write a program that will display your name on the screen. Place a comment with today’s date at the top of the program. Test your program by entering, compiling, and running it.
5. 2.5 All Java source code file names must end with .
 1. a semicolon
 2. *.class*
 3. *.java*
 4. none of the above
6. 2.6 Every Java application program must have .
 1. a method named `main`
 2. more than one class definition
 3. one or more comments

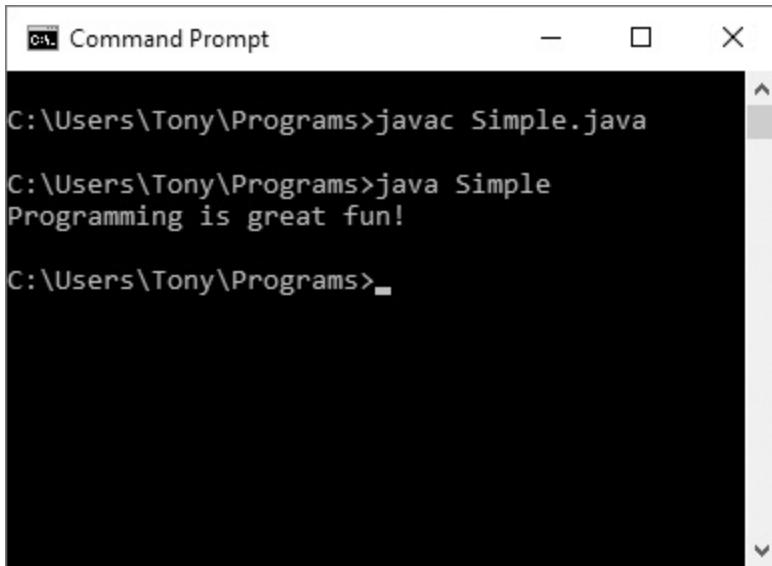
2.2 The `System.out.print` and `System.out.println` Methods, and the Java API

Concept:

The `System.out.print` and `System.out.println` methods are used to display text output. They are part of the Java API, which is a collection of prewritten classes and methods for performing specific operations.

In this section, you will learn how to write programs that produce output on the screen. The simplest type of output that a program can display on the screen is console output. *Console output* is merely plain text. When you display console output in a system that uses a graphical user interface, such as Windows or Mac OS, the output usually appears in a window similar to the one shown in [Figure 2-2](#).

Figure 2-2 A console window



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text:
C:\Users\Tony\Programs>javac Simple.java
C:\Users\Tony\Programs>java Simple
Programming is great fun!
C:\Users\Tony\Programs>

[Figure 2-2 Full Alternative Text](#)

The word *console* is an old computer term. It comes from the days when the operator of a large computer system interacted with the system by typing on a terminal that consisted of a simple screen and a keyboard. This terminal was known as the *console*. The console screen, which displayed only text, was known as the standard output device. Today, the term *standard output device* typically refers to the device that displays console output.



[VideoNote](#) Displaying Console Output

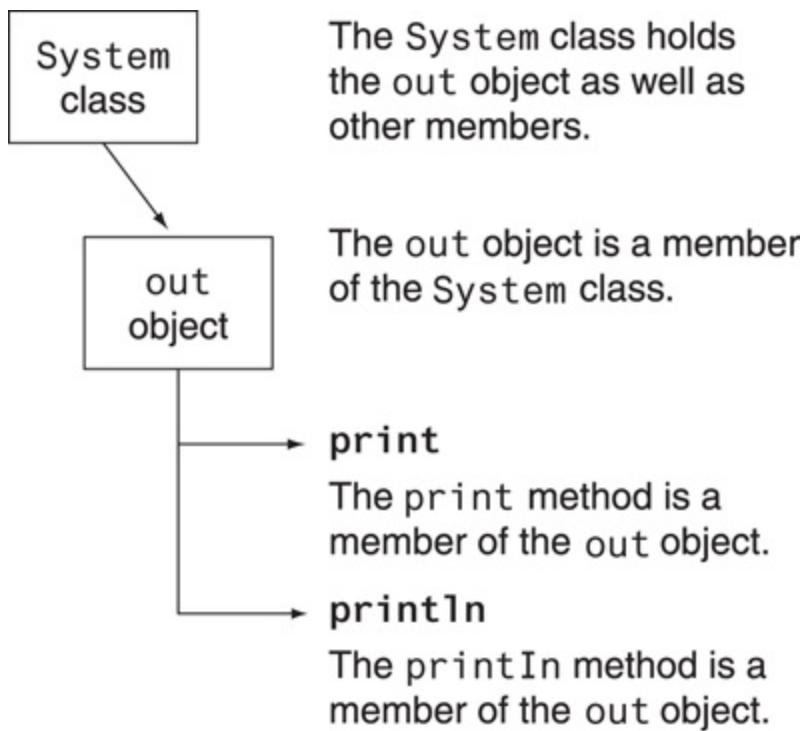
Performing output in Java, as well as many other tasks, is accomplished by using the Java API. The term API stands for *Application Programmer Interface*. The API is a standard library of prewritten classes for performing specific operations. These classes and their methods are available to all Java programs. The `print` and `println` methods are part of the API, and provide ways for output to be displayed on the standard output device.

The program in [Code Listing 2-1](#) (`Simple.java`) uses the following statement to print a message on the screen:

```
System.out.println("Programming is great fun!");
```

`System` is a class that is part of the Java API. The `System` class contains objects and methods that perform system-level operations. One of the objects contained in the `System` class is named `out`. The `out` object has methods, such as `print` and `println`, for performing output on the system console, or standard output device. The hierarchical relationship among `System`, `out`, `print`, and `println` is shown in [Figure 2-3](#).

Figure 2-3 Relationship among the `System` class, the `out` object, and the `print` and `println` methods



[Figure 2-3 Full Alternative Text](#)

Here is a brief summary of how it all works together:

- The `System` class is part of the Java API. It has member objects and methods for performing system-level operations, such as sending output to the console.
- The `out` object is a member of the `System` class. It provides methods for sending output to the screen.
- The `print` and `println` methods are members of the `out` object. They actually perform the work of writing characters on the screen.

This hierarchy explains why the statement that executes `println` is so long. The sequence `System.out.println` specifies that `println` is a member of `out`, which is a member of `System`.



Note:

The period that separates the names of the objects is pronounced “dot.” `System.out.println` is pronounced “system dot out dot print line.”

The value that is to be displayed on the screen is placed inside the parentheses. This value is known as an *argument*. For example, the following statement executes the `println` method using the string "King Arthur" as its argument. This will print "King Arthur" on the screen. (The quotation marks are not displayed on the screen.)

```
System.out.println("King Arthur");
```

An important thing to know about the `println` method is that after it displays its message, it advances the cursor to the beginning of the next line. The next item printed on the screen will begin in this position. For example, look at the program in [Code Listing 2-2](#).

Code Listing 2-2 (`TwoLines.java`)

```
1 // This is another simple Java program.
```

```
2
3 public class TwoLines
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("Programming is great fun!");
8         System.out.println("I can't get enough of it!");
9     }
10 }
```

Program Output

Programming is great fun!
I can't get enough of it!

Because each string was printed with separate `println` statements, they appear on separate lines.

The `print` Method

The `print` method, which is also part of the `System.out` object, serves a purpose similar to that of `println`—to display output on the screen. The `print` method, however, does not advance the cursor to the next line after its message is displayed. Look at [Code Listing 2-3](#).

Code Listing 2-3 (`GreatFun.java`)

```
1 // This is another simple Java program.
2
3 public class GreatFun
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("Programming is ");
8         System.out.println("great fun!");
9     }
10 }
```

Program Output

Programming is great fun!

An important concept to understand about [Code Listing 2-3](#) is that, although the output is broken up into two programming statements, this program will still display the message on one line. The data that you send to the `print` method is displayed in a continuous stream. Sometimes, this can produce less-than-desirable results. The program in [Code Listing 2-4](#) is an example.

Code Listing 2-4 (`Unruly.java`)

```
1 // An unruly printing program
2
3 public class Unruly
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("These are our top sellers:");
8         System.out.print("Computer games");
9         System.out.print("Coffee");
10        System.out.println("Aspirin");
11    }
12 }
```

Program Output

These are our top sellers:Computer gamesCoffeeAspirin

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, even though the output is broken up into four lines in the source code (lines 7 through 10), it comes out on the screen as one line. Second, notice that some of the words that are displayed are not separated by spaces. The strings are displayed exactly as they are sent to the `print` method. If spaces are to be displayed, they must appear in the strings.

There are two ways to fix this program. The most obvious way is to use `println` methods instead of `print` methods. Another way is to use escape sequences to separate the output into different lines. An *escape sequence*

starts with the backslash character (\) and is followed by one or more *control characters*. It allows you to control the way output is displayed by embedding commands within the string itself. The escape sequence that causes the output cursor to go to the next line is \n. [Code Listing 2-5](#) illustrates its use.

Code Listing 2-5 (Adjusted.java)

```
1 // A well-adjusted printing program
2
3 public class Adjusted
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("These are our top sellers:\n");
8         System.out.print("Computer games\nCoffee\n");
9         System.out.println("Aspirin");
10    }
11 }
```

Program Output

```
These are our top sellers:
Computer games
Coffee
Aspirin
```

The \n characters are called the *newline escape sequence*. When the print or println methods encounter \n in a string, they do not print the \n characters on the screen, but interpret them as a special command to advance the output cursor to the next line. There are several other escape sequences as well. For instance, \t is the *tab escape sequence*. When print or println encounters it in a string, it causes the output cursor to advance to the next tab position. [Code Listing 2-6](#) shows it in use.

Code Listing 2-6 (Tabs.java)

```
1 // Another well-adjusted printing program
```

```

2
3 public class Tabs
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("These are our top sellers:\n");
8         System.out.print("\tComputer games\n\tCoffee\n");
9         System.out.println("\tAspirin");
10    }
11 }

```

Program Output

These are our top sellers:
 Computer games
 Coffee
 Aspirin



Note:

Although you have to type two characters to write an escape sequence, they are stored in memory as a single character.

[Table 2-2](#) lists the common escape sequences and describes them.

Table 2-2 Common escape sequences

Escape Sequence	Name	Description
\n	Newline	Advances the cursor to the next line for subsequent printing
\t	Horizontal tab	Causes the cursor to skip over to the next tab stop
		Causes the cursor to back up, or move left,

\b	Backspace one position
\r	Return Causes the cursor to go to the beginning of the current line, not the next line
\\\	Backslash Causes a backslash to be printed
\'	Single quote Causes a single quotation mark to be printed
\\"	Double quote Causes a double quotation mark to be printed



Warning!

Do not confuse the backslash (\) with the forward slash (/). An escape sequence will not work if you accidentally start it with a forward slash. Also, do not put a space between the backslash and the control character.



Checkpoint

- 2.7 The following program will not compile because the lines have been mixed up.

```
System.out.print("Success\n");
}
public class Success
{
System.out.print("Success\n");
public static void main(String[] args)
System.out.print("Success ");
}
// It's a mad, mad program.
System.out.println("\nSuccess");
{
```

When the lines are arranged properly, the program should display the following output on the screen:

Program Output:

Success
Success Success

Success

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2. 2.8 Study the following program and show what it will print on the screen.

```
// The Works of Wolfgang
public class Wolfgang
{
    public static void main(String[] args)
    {
        System.out.print("The works of Wolfgang\ninclude ");
        System.out.print("the following");
        System.out.print("\nThe Turkish March ");
        System.out.print("and Symphony No. 40 ");
        System.out.println("in G minor.");
    }
}
```

3. 2.9 Write a program that will display your name on the first line; your street address on the second line; your city, state, and ZIP code on the third line; and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by compiling, and running it.

2.3 Variables and Literals

Concept:

A *variable* is a named storage location in the computer's memory. A *literal* is a value that is written into the code of a program.

As you discovered in [Chapter 1](#), variables allow you to store and work with data in the computer's memory. Part of the job of programming is to determine how many variables a program will need, and what types of data they will hold. The program in [Code Listing 2-7](#) is an example of a Java program with a variable.

Code Listing 2-7 (Variable.java)

```
1 // This program has a variable.  
2  
3 public class Variable  
4 {  
5     public static void main(String[] args)  
6     {  
7         int value;  
8  
9         value = 5;  
10        System.out.print("The value is ");  
11        System.out.println(value);  
12    }  
13 }
```

Program Output

The value is 5

Let's look more closely at this program. Here is line 7:

```
int value;
```



VideoNote Declaring Variables

This is called a variable declaration. Variables must be declared before they can be used. A variable declaration tells the compiler the variable's name and the type of data it will hold. This line indicates that the variable's name is `value`. The word `int` stands for integer, so `value` will only be used to hold integer numbers. Notice that variable declarations end with a semicolon. The next statement in this program appears in line 9:

```
value = 5;
```

This is called an assignment statement. The equal sign is an operator that stores the value on its right (in this case 5) into the variable named on its left. After this line executes, the `value` variable will contain the value 5.



Note:

This line does not print anything on the computer screen. It runs silently behind the scenes.

Now look at lines 10 and 11:

```
System.out.print("The value is ");
System.out.println(value);
```

The statement in line 10 sends the string literal "The value is" to the `print` method. The statement in line 11 sends the name of the `value` variable to the `println` method. When you send a variable name to `print` or `println`, the variable's contents are displayed. Notice there are no quotation marks around

value. Look at what happens in [Code Listing 2-8](#).

Code Listing 2-8 (Variable2.java)

```
1 // This program has a variable.  
2  
3 public class Variable2  
4 {  
5     public static void main(String[] args)  
6     {  
7         int value;  
8  
9         value = 5;  
10        System.out.print("The value is ");  
11        System.out.println("value");  
12    }  
13 }
```

Program Output

The value is value

When double quotation marks are placed around the word `value`, it becomes a string literal, not a variable name. When string literals are sent to `print` or `println`, they are displayed exactly as they appear inside the quotation marks.

Displaying Multiple Items with the + Operator

When the `+` operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will print:

```
This is one string.
```

The + operator produces a string that is the combination of the two strings used as its operands. You can also use the + operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;  
System.out.println("The value is " + number);
```

The second line uses the + operator to concatenate the contents of the number variable with the string “The value is”. Although number is not a string, the + operator converts its value to a string then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Sometimes, the argument you use with print or println is too long to fit on one line in your program code. However, a string literal cannot begin on one line and end on another. For example, the following will cause an error:

```
// This is an error!  
System.out.println("Enter a value that is greater than zero and 1
```

You can remedy this problem by breaking the argument up into smaller string literals and then using the string concatenation operator to spread them out over more than one line. Here is an example:

```
System.out.println("Enter a value that is " +  
    "greater than zero and less " +  
    "than 10.");
```

In this statement, the argument is broken up into three strings and joined using the + operator. The following example shows the same technique used when the contents of a variable are part of the concatenation:

```
sum = 249;  
System.out.println("The sum of the three "  
    "numbers is " + sum);
```

Be Careful with Quotation Marks

As shown in [Code Listing 2-8](#), placing quotation marks around a variable name changes the program’s results. In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in [Code Listings 2-7](#) and [2-8](#), the number 5 was assigned to the variable `value`. It would have been an error to perform the assignment this way:

```
value = "5";    // Error!
```

In this statement, 5 is no longer an integer, but a string literal. Because `value` was declared an integer variable, you can only store integers in it. In other words, 5 and “5” are not the same thing.

The fact that numbers can be represented as strings frequently confuses students who are new to programming. Just remember that strings are intended for humans to read. They are to be displayed on computer screens or printed on paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings, and before numbers can be displayed on the screen, first they must be converted to strings. (Fortunately, `print` and `println` handle the conversion automatically when you send numbers to them.) Don’t fret if this still bothers you. Later in this chapter, we will shed more light on the differences among numbers, characters, and strings by discussing their internal storage.

More about Literals

A literal is a value that is written in the code of a program. Literals are commonly assigned to variables or displayed. [Code Listing 2-9](#) contains both literals and a variable.

Code Listing 2-9 (`Literals.java`)

```

1 // This program has literals and a variable.
2
3 public class Literals
4 {
5     public static void main(String[] args)
6     {
7         int apples;
8
9         apples = 20;
10        System.out.println("Today we sold " + apples +
11                            " bushels of apples.");
12    }
13 }
```

Program Output

Today we sold 20 bushels of apples.

Of course, the variable in this program is apples. It is declared as an integer. [Table 2-3](#) shows a list of the literals found in the program.

Table 2-3 Literals

Literal	Type of Literal
20	Integer literal
"Today we sold "	String literal
" bushels of apples."	String literal

Identifiers

An *identifier* is a programmer-defined name that represents some element of a program. Variable names and class names are examples of identifiers. You may choose your own variable names and class names in Java, as long as you do not use any of the Java key words. The *key words* make up the core of the language, and each has a specific purpose. [Table 1-3](#) in [Chapter 1](#) and Appendix C show a complete list of Java key words.

You should always choose names for your variables that give an indication of what they are used for. You may be tempted to declare variables with names like this:

```
int x;
```

The rather nondescript name, `x`, gives no clue as to what the variable's purpose is. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea for what the variable is used. This method of coding helps produce *self-documenting programs*, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines of code, it is important that they be as self-documenting as possible.

You have probably noticed the mixture of uppercase and lowercase letters in the name `itemsOrdered`. Although all of Java's key words must be written in lowercase, you may use uppercase letters in variable names. The reason the `o` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is used as two words. Variable names cannot contain spaces, however, so the two words must be combined. When "items" and "ordered" are stuck together, you get a variable declaration like this:

```
int itemsOrdered;
```

Capitalization of the letter `O` makes `itemsOrdered` easier to read. Typically, variable names begin with a lowercase letter, and after that, the first letter of each individual word that makes up the variable name is capitalized.

The following are some specific rules that must be followed with all identifiers:

- The first character must be one of the letters `a–z`, `A–Z`, an underscore (`_`), or a dollar sign (`$`).
- After the first character, you may use the letters `a–z` or `A–Z`, the digits

0–9, underscores (_), or dollar signs (\$).

- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.



Note:

Although the \$ is a legal identifier character, it is normally used for special purposes. So, don't use it in your variable names.

[Table 2-4](#) shows a list of variable names, and tells if each is legal or illegal in Java.

Table 2-4 Some variable names

Variable Name	Legal or Illegal?
<code>dayOfWeek</code>	Legal
<code>3dGraph</code>	Illegal because identifiers cannot begin with a digit
<code>june1997</code>	Legal
<code>mixture#3</code>	Illegal because identifiers may only use alphabetic letters, digits, underscores, or dollar signs
<code>week day</code>	Illegal because identifiers cannot contain spaces

Class Names

As mentioned before, it is standard practice to begin variable names with a lowercase letter, then capitalize the first letter of each subsequent word that makes up the name. It is also a standard practice to capitalize the first letter of

a class name, as well as the first letter of each subsequent word it contains. This helps differentiate the names of variables from the names of classes. For example, payRate would be a variable name, and Employee would be a class name.



Checkpoint

1. 2.10 Examine the following program.

```
// This program uses variables and literals.

public class BigLittle
{
    public static void main(String[] args)
    {
        int little;
        int big;

        little = 2;
        big = 2000;
        System.out.println("The little number is " + little);
        System.out.println("The big number is " + big);
    }
}
```

List the variables and literals found in the program.

2. 2.11 What will the following program display on the screen?

```
public class CheckPoint
{
    public static void main(String[] args)
    {
        int number;

        number = 712;
        System.out.println("The value is " + "number");
    }
}
```

2.4 Primitive Data Types

Concept:

There are many different types of data. Variables are classified according to their data type, which determines the kind of data that may be stored in them.

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric data, for example, there are whole and fractional numbers; negative and positive numbers; and numbers so large and others so small that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as strings of characters. When you write a program, you must determine what types of data it is likely to encounter.

Each variable has a *data type*, which is the type of data that the variable can hold. Selecting the proper data type is important because a variable's data type determines the amount of memory the variable uses, and the way the variable formats and stores data. It is important to select a data type that is appropriate for the type of data with which your program will work. If you are writing a program to calculate the number of miles to a distant star, you need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you need variables that store very small and precise numbers. If you are writing a program that must perform thousands of intensive calculations, you want variables that can be processed quickly. The data type of a variable determines all of these factors.

[Table 2-5](#) shows all of the Java *primitive data types* for holding numeric data.

Table 2-5 Primitive data types

for numeric data

Data Type	Size	Range
byte	1 byte	Integers in the range of -128 to +127
short	2 bytes	Integers in the range of -32,768 to +32,767
int	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
long	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4 bytes	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy
double	8 bytes	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy

The words listed in the left column of [Table 2-5](#) are the key words you use in variable declarations. A variable declaration takes the following general format:

DataType *VariableName*;

DataType is the name of the data type, and *VariableName* is the name of the variable. Here are some examples of variable declarations:

```
byte inches;  
int speed;  
short month;  
float salesCommission;  
double distance;
```

The size column in [Table 2-5](#) shows the number of bytes that a variable of each of the data types uses. For example, an *int* variable uses 4 bytes, and a *double* variable uses 8 bytes. The range column shows the ranges of numbers that may be stored in variables of each data type. For example, an *int*

variable can hold numbers from -2,147,483,648 up to +2,147,483,647. One of the appealing characteristics of the Java language is that the sizes and ranges of all the primitive data types are the same on all computers.



Note:

These data types are called “primitive” because you cannot use them to create objects. Recall from [Chapter 1](#)’s discussion on object-oriented programming that an object has attributes and methods. With the primitive data types, you can only create variables, and a variable can only be used to hold a single value. Such variables do not have attributes or methods.

The Integer Data Types

The first four data types listed in [Table 2-5](#), byte, int, short, and long, are all integer data types. An integer variable can hold whole numbers such as 7, 125, -14, and 6928. The program in [Code Listing 2-10](#) shows several variables of different integer data types being used.

Code Listing 2-10 (IntegerVariables.java)

```
1 // This program has variables of several of the integer types.  
2  
3 public class IntegerVariables  
4 {  
5     public static void main(String[] args)  
6     {  
7         int checking; // Declare an int variable named checking.  
8         byte miles; // Declare a byte variable named miles.  
9         short minutes; // Declare a short variable named minutes.  
10        long diameter; // Declare a long variable named diameter.  
11  
12        checking = -20;
```

```
13     miles = 105;
14     minutes = 120;
15     diameter = 100000;
16     System.out.println("We have made a journey of " + miles +
17             " miles.");
18     System.out.println("It took us " + minutes + " minutes.");
19     System.out.println("Our account balance is $" + checking);
20     System.out.println("The galaxy is " + diameter +
21             " light years in diameter.");
22 }
23 }
```

Program Output

We have made a journey of 105 miles.
It took us 120 minutes.
Our account balance is \$-20
The galaxy is 100000 light years in diameter.

In most programs, you will need more than one variable of any given data type. If a program uses three integers, length, width, and area, they could be declared separately, as follows:

```
int length;
int width;
int area;
```

It is easier, however, to combine the three variable declarations:

```
int length, width, area;
```

You can declare several variables of the same type, simply by separating their names with commas and spaces.

Integer Literals

When you write an integer literal in your program code, Java assumes it to be of the `int` data type. For example, in [Code Listing 2-10](#), the literals -20, 105, 120, and 189000 are all treated as `int` values. You can force an integer literal to be treated as a `long`, however, by suffixing it with the letter L. For

example, the value 57L would be treated as a long. You can use either an uppercase or lowercase L. The lowercase l looks too much like the number 1, so you should always use the uppercase L.



Warning!

You cannot embed commas in numeric literals. For example, the following statement will cause an error:

```
number = 1,257,649;      // ERROR!
```

This statement must be written as:

```
number = 1257649;      // Correct.
```

Floating-Point Data Types

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers. Values such as 1.7 and -45.316 are floating-point numbers.

In Java, there are two data types that can represent floating-point numbers. They are `float` and `double`. The `float` data type is considered a single-precision data type. It can store a floating-point number with 7 digits of accuracy. The `double` data type is considered a double-precision data type. It can store a floating-point number with 15 digits of accuracy. The `double` data type uses twice as much memory as the `float` data type, however. A `float` variable occupies 4 bytes of memory, whereas a `double` variable uses 8 bytes.

[Code Listing 2-11](#) shows a program that uses three `double` variables.

Code Listing 2-11 (`Sale.java`)

```
1 // This program demonstrates the double data type.  
2  
3 public class Sale  
4 {  
5     public static void main(String[] args)  
6     {  
7         double price, tax, total;  
8  
9         price = 29.75;  
10        tax = 1.76;  
11        total = 31.51;  
12        System.out.println("The price of the item " +  
13                            "is " + price);  
14        System.out.println("The tax is " + tax);  
15        System.out.println("The total is " + total);  
16    }  
17 }
```

Program Output

```
The price of the item is 29.75  
The tax is 1.76  
The total is 31.51
```

Floating-Point Literals

When you write a floating-point literal in your program code, Java assumes it to be of the double data type. For example, in [Code Listing 2-11](#), the literals 29.75, 1.76, and 31.51 are all treated as double values. Because of this, a problem can arise when assigning a floating-point literal to a float variable. Java is a *strongly typed language*, which means that it only allows you to store values of compatible data types in variables. A double value is not compatible with a float variable, because a double can be much larger or much smaller than the allowable range for a float. As a result, code such as the following will cause an error:

```
float number;  
number = 23.5;      // Error!
```

You can force a double literal to be treated as a float, however, by suffixing it with the letter F or f. The preceding code can be rewritten in the following

manner to prevent an error:

```
float number;  
number = 23.5F;      // This will work.
```



Warning!

If you are working with literals that represent dollar amounts, remember that you cannot embed currency symbols (such as \$) or commas in the literal. For example, the following statement will cause an error:

```
grossPay = $1,257.00;    // ERROR!
```

This statement must be written as:

```
grossPay = 1257.00;    // Correct.
```

Scientific and E Notation

Floating-point literals can be represented in scientific notation. Take the number 47,281.97. In scientific notation, this number is 4.728197×10^4 . (10^4 is equal to 10,000, and $4.728197 \times 10,000$ is 47,281.97.)

Java uses E notation to represent values in scientific notation. In E notation, the number 4.728197×10^4 would be 4.728197E4. [Table 2-6](#) shows other numbers represented in scientific and E notation.

Table 2-6 Floating-point representations

Decimal Notation Scientific Notation E Notation

247.91	2.4791×10^2	2.4791E2
--------	----------------------	----------

0.00072	7.2×10^{-4}	7.2E-4
2 900 000	2.9×10^6	2.9E6



Note:

The E can be uppercase or lowercase.

[Code Listing 2-12](#) demonstrates the use of floating-point literals expressed in E notation.

Code Listing 2-12 (SunFacts.java)

```

1 // This program uses E notation.
2
3 public class SunFacts
4 {
5     public static void main(String[] args)
6     {
7         double distance, mass;
8
9         distance = 1.495979E11;
10        mass = 1.989E30;
11        System.out.println("The Sun is " + distance +
12                            "meters away.");
13        System.out.println("The Sun's mass is " + mass +
14                            "kilograms.");
15    }
16 }
```

Program Output

The Sun is 1.495979E11 meters away.
 The Sun's mass is 1.989E30 kilograms.

The boolean Data Type

The boolean data type allows you to create variables that may hold one of two possible values: true or false. [Code Listing 2-13](#) demonstrates the declaration and assignment of a boolean variable.

Code Listing 2-13 (TrueFalse.java)

```
1 // A program for demonstrating boolean variables
2
3 public class TrueFalse
4 {
5     public static void main(String[] args)
6     {
7         boolean bool;
8
9         bool = true;
10        System.out.println(bool);
11        bool = false;
12        System.out.println(bool);
13    }
14 }
```

Program Output

```
true
false
```

Variables of the boolean data type are useful for evaluating conditions that are either true or false. You will not be using them until [Chapter 3](#), so for now, just remember the following things:

- boolean variables may only hold the values true or false.
- The contents of a boolean variable may not be assigned to a variable of any type other than boolean.

The char Data Type

The char data type is used to store characters. A variable of the char data

type can hold one character at a time. Character literals are enclosed in *single quotation marks*. The program in [Code Listing 2-14](#) uses a char variable. While the program runs, the character literals ‘A’ and ‘B’ are assigned to the variable.

Code Listing 2-14 (Letters.java)

```
1 // This program demonstrates the char data type.  
2  
3 public class Letters  
4 {  
5     public static void main(String[] args)  
6     {  
7         char letter;  
8  
9         letter = 'A';  
10        System.out.println(letter);  
11        letter = 'B';  
12        System.out.println(letter);  
13    }  
14 }
```

Program Output

A
B

It is important that you do not confuse character literals with string literals, which are enclosed in double quotation marks. String literals cannot be assigned to char variables.

Unicode

Characters are internally represented by numbers. Each printable character, as well as many nonprintable characters, is assigned a unique number. Java uses Unicode, which is a set of numbers that are used as codes for representing characters. Each Unicode number requires two bytes of memory, so char variables occupy two bytes. When a character is stored in memory, it is

actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

You may want to refer to [Appendix A](#), which shows a portion of the Unicode character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. [Code Listing 2-15](#) demonstrates that when you work with characters, you are actually working with numbers.

Code Listing 2-15 (Letters2.java)

```
1 // This program demonstrates the close relationship between
2 // characters and integers.
3
4 public class Letters2
5 {
6     public static void main(String[] args)
7     {
8         char letter;
9
10        letter = 65;
11        System.out.println(letter);
12        letter = 66;
13        System.out.println(letter);
14    }
15 }
```

Program Output

A
B

[Figure 2-4](#) illustrates that when you think of the characters A, B, and C being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

Figure 2-4 Characters and how they are stored in memory

A

B

C

These characters are stored in memory as...

00	65	00	66	00	67
----	----	----	----	----	----

[Figure 2-4 Full Alternative Text](#)

Variable Assignment and Initialization

As you have already seen in several examples, a value is put into a variable with an *assignment statement*. For example, the following statement assigns the value 12 to the variable `unitsSold`:

```
unitsSold = 12;
```

The `=` symbol is called the assignment operator. Operators perform operations on data. The data that operators work with are called operands. The assignment operator has two operands. In the above statement, the operands are `unitsSold` and 12.

In an assignment statement, the name of the variable receiving the assignment must appear on the left side of the operator, and the value being assigned must appear on the right side. The following statement is incorrect:

```
12 = unitsSold; // ERROR!
```

The operand on the left side of the `=` operator must be a variable name. The operand on the right side of the `=` symbol must be an expression that has a value. The assignment operator takes the value of the right operand and puts it in the variable identified by the left operand. Assuming that `length` and `width` are both `int` variables, the following code illustrates that the assignment operator's right operand may be a literal or a variable:

```
length = 20;  
width = length;
```

It is important to note that the assignment operator only changes the contents of its left operand. The second statement assigns the value of the `length` variable to the `width` variable. After the statement has executed, `length` still has the same value, 20.

You may also assign values to variables as part of the declaration statement. This is known as *initialization*. [Code Listing 2-16](#) shows how this is done.

Code Listing 2-16 (Initialize.java)

```
1 // This program shows variable initialization.  
2  
3 public class Initialize  
4 {  
5     public static void main(String[] args)  
6     {  
7         int month = 2, days = 28;  
8  
9         System.out.println("Month " + month + " has " +  
10                days + " days.");  
11    }  
12 }
```

Program Output

Month 2 has 28 days.

The variable declaration statement in this program is in line 7:

```
int month = 2, days = 28;
```

This statement declares the `month` variable and initializes it with the value 2, and declares the `days` variable and initializes it with the value 28. As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other declaration

statements that perform initialization:

```
double payRate = 25.52;
float interestRate = 12.9F;
char stockCode = 'D';
int customerNum = 459;
```

Of course, there are always variations on a theme. Java allows you to declare several variables and only initialize some of them. Here is an example of such a declaration:

```
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89, and `departure` is initialized to 10. The `travelTime` and `distance` variables remain uninitialized.



Warning!

When a variable is declared inside a method, it must have a value stored in it before it can be used. If the compiler determines that the program might be using such a variable before a value has been stored in it, an error will occur. You can avoid this type of error by initializing the variable with a value.

Variables Hold Only One Value at a Time

Remember, a variable can hold only one value at a time. When you assign a new value to a variable, the new value takes the place of the variable's previous contents. For example, look at the following code.

```
int x = 5;
System.out.println(x);
x = 99;
System.out.println(x);
```

In this code, the variable `x` is initialized with the value 5 and its contents are displayed. Then the variable is assigned the value 99. This value overwrites the value 5 that was previously stored there. The code will produce the following output:

```
5  
99
```



Checkpoint

1. 2.12 Which of the following are illegal variable names and why?

```
x  
99bottles  
july97  
theSalesFigureForFiscalYear98  
r&d  
grade_report
```

2. 2.13 Is the variable name `Sales` the same as `sales`? Why or why not?
3. 2.14 Refer to the Java primitive data types listed in [Table 2-5](#) for this question.
 1. If a variable needs to hold whole numbers in the range 32 to 6,000, what primitive data type would be best?
 2. If a variable needs to hold whole numbers in the range -40,000 to +40,000, what primitive data type would be best?
 3. Which of the following literals use more memory? `22.1` or `22.1F`?
4. 2.15 How would the number 6.31×10^{17} be represented in E notation?
5. 2.16 A program declares a `float` variable named `number`, and the following statement causes an error. What can be done to fix the error?

```
number = 7.4;
```

6. 2.17 What values can boolean variables hold?
7. 2.18 Write statements that do the following:
 1. Declare a char variable named letter.
 2. Assign the letter A to the letter variable.
 3. Display the contents of the letter variable.
8. 2.19 What are the Unicode codes for the characters 'c', 'F', and 'w'? (You may need to refer to [Appendix A](#).)
9. 2.20 Which is a character literal, ‘B’ or “B”?
10. 2.21 What is wrong with the following statement?

```
char letter = "Z";
```

2.5 Arithmetic Operators

Concept:

There are many operators for manipulating numeric values and performing arithmetic operations.

Java offers a multitude of operators for manipulating data. Generally, there are three types of operators: *unary*, *binary*, and *ternary*. These terms reflect the number of operands an operator requires.



VideoNote Simple Math Expressions

Unary operators require only a single operand. For example, consider the following expression:

-5

Of course, we understand this represents the value negative five. We can also apply the operator to a variable, as follows:

-number

This expression gives the negative of the value stored in `number`. The minus sign, when used this way, is called the *negation operator*. Because it only requires one operand, it is a unary operator.

Binary operators work with two operands. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. Java has only one ternary operator, which will be discussed in [Chapter 4](#).

Arithmetic operations are very common in programming. [Table 2-7](#) shows the arithmetic operators in Java.

Table 2-7 Arithmetic operators

Operator	Meaning	Type	Example
+	Addition	Binary	total = cost + tax;
-	Subtraction	Binary	cost = total - tax;
*	Multiplication	Binary	tax = cost * rate;
/	Division	Binary	salePrice = original / 2;
%	Modulus	Binary	remainder = value % 3;

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. Here are some example statements that use the addition operator:

```
amount = 4 + 8;           // Assigns 12 to amount
total = price + tax;      // Assigns price + tax to total
number = number + 1;       // Assigns number + 1 to number
```

The subtraction operator returns the value of its right operand subtracted from its left operand. Here are some examples:

```
temperature = 112 - 14;    // Assigns 98 to temperature
sale = price - discount;  // Assigns price - discount to sale
number = number - 1;       // Assigns number - 1 to number
```

The multiplication operator returns the product of its two operands. Here are some examples:

```
markUp = 12 * 0.25;        // Assigns 3 to markUp
commission = sales * percent; // Assigns sales * percent to co
population = population * 2; // Assigns population * 2 to pop
```

The division operator returns the quotient of its left operand divided by its right operand. Here are some examples:

```
points = 100 / 20;         // Assigns 5 to points
```

```
teams = players / maxEach; // Assigns players / maxEach to teams
half = number / 2; // Assigns number / 2 to half
```

The modulus operator returns the remainder of a division operation involving two integers. The following statement assigns 2 to leftOver:

```
leftOver = 17 % 3;
```

Situations will arise where you need to get the remainder of a division. Computations that detect odd numbers, or are required to determine how many items are left over after division, use the modulus operator.

The program in [Code Listing 2-17](#) demonstrates some of these operators used in a simple payroll calculation.

Code Listing 2-17 (Wages.java)

```
1 // This program calculates hourly wages plus overtime.
2
3 public class Wages
4 {
5     public static void main(String[] args)
6     {
7         double regularWages; // The calculated regular wages
8         double basePay = 25; // The base pay rate
9         double regularHours = 40; // The hours worked less overtime
10        double overtimeWages; // Overtime wages
11        double overtimePay = 37.5; // Overtime pay rate
12        double overtimeHours = 10; // Overtime hours worked
13        double totalWages; // Total wages
14
15        regularWages = basePay * regularHours;
16        overtimeWages = overtimePay * overtimeHours;
17        totalWages = regularWages + overtimeWages;
18        System.out.println("Wages for this week are $" +
19                           totalWages);
20    }
21 }
```

Program Output

Wages for this week are \$1375.0

[Code Listing 2-17](#) calculates the total wages an hourly paid worker earned in one week. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Line 15 in the program multiplies `basePay` times `regularHours` and stores the result, which is 1000, in `regularWages`:

```
regularWages = basePay * regularHours;
```

Line 16 multiplies `overtimePay` times `overtimeHours` and stores the result, which is 375, in `overtimeWages`:

```
overtimeWages = overtimePay * overtimeHours;
```

Line 17 adds the regular wages and the overtime wages and stores the result, 1375, in `totalWages`:

```
totalWages = regularWages + overtimeWages;
```

The `println` statement in lines 18 and 19 displays the message on the screen reporting the week's wages.

Integer Division

When both operands of the division operator are integers, the operator will perform *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;  
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the `number` variable. What will be stored in `number`? You would probably assume that 2.5 would be stored in `number` because that is the result your calculator shows when you

divide 5 by 2. However, that is not what happens when the previous Java code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or *truncated*. As a result, the value 2 will be assigned to the number variable.

In the previous code, it doesn't matter that number is declared as a double because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;  
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

Operator Precedence

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, x, 21, and y to the variable answer:

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in outcome? The 6 is used as an operand for both the addition and division operators. The outcome variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the division operator has higher *precedence* than the addition operator.

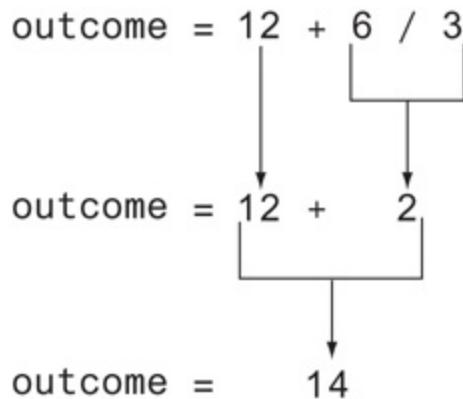
Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence works

first. Multiplication and division have higher precedence than addition and subtraction, so the statement above works like this:

1. 6 is divided by 3, yielding a result of 2
2. 12 is added to 2, yielding a result of 14

This could be diagrammed as shown in [Figure 2-5](#).

Figure 2-5 Precedence illustrated



[Figure 2-5 Full Alternative Text](#)

[Table 2-8](#) shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below it.

Table 2-8 Precedence of arithmetic operators (highest to lowest)

Highest Precedence → - (unary negation)

* / %

Lowest Precedence → + -

The multiplication, division, and modulus operators have the same precedence. The addition and subtraction operators have the same precedence. If two operators sharing an operand have the same precedence, they work according to their *associativity*. Associativity is either *left to right* or *right to left*. [Table 2-9](#) shows the arithmetic operators and their associativity.

Table 2-9 Associativity of arithmetic operators

Operator	Associativity
- (unary negation)	Right to left
* / %	Left to right
+ -	Left to right

[Table 2-10](#) shows some expressions and their values.

Table 2-10 Some expressions and their values

Expression	Value
5 + 2 * 4	13
10 / 2 - 3	2
8 + 12 * 2 - 4	28
4 + 17 % 2 - 1	4
6 - 3 * 2 + 7 - 16	

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the statement below, the sum of a, b, c, and d is divided by 4.0.

```
average = (a + b + c + d) / 4.0;
```

Without the parentheses, however, d would be divided by 4 and the result added to the sum of a, b, and c. [Table 2-11](#) shows more expressions and their values.

Table 2-11 More expressions and their values

Expression	Value
(5 + 2) * 4	28
10 / (5 - 3)	5
8 + 12 * (6 - 2)	56
(4 + 17) % 2 - 1	0
(6 - 3) * (2 + 7) / 39	

In the Spotlight: Calculating Percentages and Discounts



Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including Java) do not use the %

symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent is written as 0.5 and 2 percent is written as 0.02.

Let's look at an example. Suppose you earn \$6,000 per month, and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 8 percent, or 10 percent of your gross wages. To make this determination, you write a program like the one shown in [Code Listing 2-18](#).

Code Listing 2-18 (Contribution.java)

```
1 // This program calculates the amount of pay that
2 // will be contributed to a retirement plan if 5%,
3 // 7%, or 10% of monthly pay is withheld.
4
5 public class Contribution
6 {
7     public static void main(String[] args)
8     {
9         // Variables to hold the monthly pay and
10        // the amount of contribution.
11        double monthlyPay = 6000.0;
12        double contribution;
13
14        // Calculate and display a 5% contribution.
15        contribution = monthlyPay * 0.05;
16        System.out.println("5 percent is $" +
17                            contribution +
18                            " per month.");
19
20        // Calculate and display a 8% contribution.
21        contribution = monthlyPay * 0.08;
22        System.out.println("8 percent is $" +
23                            contribution +
24                            " per month.");
25
26        // Calculate and display a 10% contribution.
27        contribution = monthlyPay * 0.1;
```

```
28     System.out.println("10 percent is $" +  
29             contribution +  
30             " per month.");  
31 }  
32 }
```

Program Output

```
5 percent is $300.0 per month.  
8 percent is $480.0 per month.  
10 percent is $600.0 per month.
```

Lines 11 and 12 declare two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value `6000.0`, holds the amount of your monthly pay. The `contribution` variable holds the amount of a contribution to the retirement plan.

The statements in lines 15 through 18 calculate and display 5 percent of the monthly pay. The calculation is done in line 15, where the `monthlyPay` variable is multiplied by `0.05`. The result is assigned to the `contribution` variable, which is then displayed by the statement in lines 16 through 18.

Similar steps are taken in lines 21 through 24, which calculate and display 8 percent of the monthly pay, and lines 27 through 30, which calculate and display 10 percent of the monthly pay.

Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at `$59`, and is planning to have a sale during which the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price, you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.

- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

[Code Listing 2-19](#) shows how this is done in Java.

Code Listing 2-19 (Discount.java)

```

1 // This program calculates the sale price of an
2 // item that is regularly priced at $59, with
3 // a 20 percent discount subtracted.
4
5 public class Discount
6 {
7     public static void main(String[] args)
8     {
9         // Variables to hold the regular price, the
10        // amount of a discount, and the sale price.
11        double regularPrice = 59.0;
12        double discount;
13        double salePrice;
14
15        // Calculate the amount of a 20% discount.
16        discount = regularPrice * 0.2;
17
18        // Calculate the sale price by subtracting
19        // the discount from the regular price.
20        salePrice = regularPrice - discount;
21
22        // Display the results.
23        System.out.println("Regular price: $" + regularPrice);
24        System.out.println("Discount amount $" + discount);
25        System.out.println("Sale price: $" + salePrice);
26    }
27 }
```

Program Output

```

Regular price: $59.0
Discount amount $11.8
Sale price: $47.2
```

Lines 11 through 13 declare three variables. The regularPrice variable

holds the item's regular price and is initialized with the value 59.0. The discount variable holds the amount of the discount once it is calculated. The salePrice variable holds the item's sale price.

Line 16 calculates the amount of the 20 percent discount by multiplying regularPrice by 0.2. The result is stored in the discount variable. Line 20 calculates the sale price by subtracting discount from regularPrice. The result is stored in the salePrice variable. The statements in lines 23 through 25 display the item's regular price, the amount of the discount, and the sale price.

The Math Class

The Java API provides a class named `Math`, which contains numerous methods that are useful for performing complex mathematical operations. In this section, we will briefly look at the `Math.pow` and `Math.sqrt` methods.

The `Math.pow` Method

In Java, raising a number to a power requires the `Math.pow` method. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments. It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

```
result = 42
```

Here is another example of a statement using the `Math.pow` method. It assigns 3 times 6³ to `x`:

```
x = 3 * Math.pow(6.0, 3.0);
```

And the following statement displays the value of 5 raised to the power of 4:

```
System.out.println(Math.pow(5.0, 4.0));
```

The Math.sqrt Method

The `Math.sqrt` method accepts a double value as its argument, and returns the square root of the value. Here is an example of how the method is used:

```
result = Math.sqrt(9.0);
```

In this example, the value 9.0 is passed as an argument to the `Math.sqrt` method. The method will return the square root of 9.0, which is assigned to the `result` variable. The following statement shows another example in which the square root of 25.0 (which is 5.0) is displayed on the screen:

```
System.out.println(Math.sqrt(25.0));
```

For more information about the `Math` class, see Appendix F, available on this book's online resource page at www.pearsonhighered.com/gaddis.



Checkpoint

- 2.22 Complete the following table by writing the value of each expression in the Value column.

Expression	Value
<code>6 + 3 * 5</code>	
<code>12 / 2 - 4</code>	
<code>9 + 14 * 2 - 6</code>	
<code>5 + 19 % 3 - 1</code>	
<code>(6 + 2) * 3</code>	
<code>14 / (11 - 4)</code>	
<code>9 + 12 * (8 - 3)</code>	

2. 2.23 Is the division statement in the following code an example of integer division or floating-point division? What value will be stored in portion?

```
double portion;  
portion = 70 / 3;
```

2.6 Combined Assignment Operators

Concept:

The combined assignment operators combine the assignment operator with the arithmetic operators.

Quite often, programs have assignment statements of the following form:

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

On the right side of the assignment operator, 1 is added to x. The result is then assigned to x, replacing the value that was previously there. Effectively, this statement adds 1 to x. Here is another example:

```
balance = balance + deposit;
```

Assuming that balance and deposit are variables, this statement assigns the value of balance + deposit to balance. The effect of this statement is that deposit is added to the value stored in balance. Here is another example:

```
balance = balance - withdrawal;
```

Assuming that balance and withdrawal are variables, this statement assigns the value of balance - withdrawal to balance. The effect of this statement is that withdrawal is subtracted from the value stored in balance.

If you have not seen these types of statements before, they might cause some initial confusion because the same variable name appears on both sides of the assignment operator. [Table 2-12](#) shows other examples of statements written this way.

Table 2-12 Various assignment statements (assume x = 6 in each statement)

Statement	What It Does	Value of x after the Statement
x = x + 4;	Adds 4 to x	10
x = x - 3;	Subtracts 3 from x	3
x = x * 10;	Multiplies x by 10	60
x = x / 2;	Divides x by 2	3
x = x % 4	Assigns the remainder of x / 4 to x.	2

These types of operations are common in programming. For convenience, Java offers a special set of operators designed specifically for these jobs. [Table 2-13](#) shows the *combined assignment operators*, also known as *compound operators*.

Table 2-13 Combined assignment operators

Operator Example Usage Equivalent to

+=	x += 5;	x = x + 5;
-=	y -= 2;	y = y - 2;
*=	z *= 10;	z = z * 10;
/=	a /= b;	a = a / b;
%=	c %= 3;	c = c % 3;

As you can see, the combined assignment operators do not require the programmer to type the variable name twice. The following statement

```
balance = balance + deposit;
```

could be rewritten as

```
balance += deposit;
```

Similarly, the statement

```
balance = balance - withdrawal;
```

could be rewritten as

```
balance -= withdrawal;
```



Checkpoint

1. 2.24 Write statements using combined assignment operators to perform the following:
 1. Add 6 to x
 2. Subtract 4 from $amount$
 3. Multiply y by 4
 4. Divide $total$ by 27
 5. Store in x the remainder of x divided by 7

2.7 Conversion between Primitive Data Types

Concept:

Before a value can be stored in a variable, the value's data type must be compatible with the variable's data type. Java performs some conversions between data types automatically, but does not automatically perform any conversion that can result in the loss of data. Java also follows a set of rules when evaluating arithmetic expressions containing mixed data types.

Java is a *strongly typed* language. This means that before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine if they are compatible. For example, look at the following statements:

```
int x;  
double y = 2.5;  
x = y;
```

The assignment statement is attempting to store a `double` value (2.5) in an `int` variable. When the Java compiler encounters this line of code, it will respond with an error message. (The Java compiler displays the message “possible loss of precision.”)

Not all assignment statements that mix data types are rejected by the compiler, however. For instance, look at the following program segment:

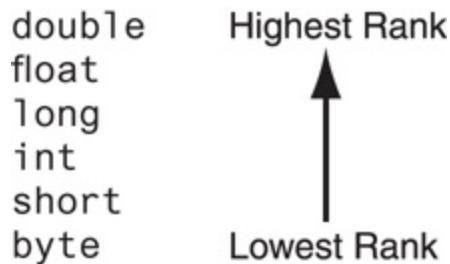
```
int x;  
short y = 2;  
x = y;
```

This assignment statement, which stores a `short` in an `int`, will work with no

problems. So why does Java permit a `short` to be stored in an `int`, but does not permit a `double` to be stored in an `int`? The reason is that a `double` can store fractional numbers and can hold values much larger than an `int` can hold. If Java were to permit a `double` to be assigned to an `int`, a loss of data would be likely.

Just like officers in the military, the primitive data types are ranked. One data type outranks another if it can hold a larger number. For example, a `float` outranks an `int`, and an `int` outranks a `short`. [Figure 2-6](#) shows the numeric data types in order of their rank. The higher a data type appears in the list, the higher is its rank.

Figure 2-6 Primitive data type ranking



[Figure 2-6 Full Alternative Text](#)

In assignment statements where values of lower-ranked data types are stored in variables of higher-ranked data types, Java automatically converts the lower-ranked value to the higher-ranked type. This is called a *widening conversion*. For example, the following code demonstrates a widening conversion, which takes place when an `int` value is stored in a `double` variable:

```
double x;  
int y = 10;  
x = y;           // Performs a widening conversion
```

A *narrowing conversion* is the conversion of a value to a lower-ranked type.

For example, converting a double to an `int` would be a narrowing conversion. Because narrowing conversions can potentially cause a loss of data, Java does not automatically perform them.

Cast Operators

The *cast operator* lets you manually convert a value, even if it means that a narrowing conversion will take place. Cast operators are unary operators that appear as a data type name enclosed in a set of parentheses. The operator precedes the value being converted. Here is an example:

```
x = (int)number;
```

The cast operator in this statement is the word `int` inside the parentheses. It returns the value in `number`, converted to an `int`. This converted value is then stored in `x`. If `number` were a floating-point variable, such as a `float` or a `double`, the value that is returned would be *truncated*, which means the fractional part of the number is lost. The original value in the `number` variable is not changed, however.

[Table 2-14](#) shows several statements using a cast operator.

Table 2-14 Example uses of cast operators

Statement	Description
<code>littleNum = (short)bigNum;</code>	The cast operator returns the value in <code>bigNum</code> , converted to a <code>short</code> . The converted value is assigned to the variable <code>littleNum</code> .
<code>x = (long)3.7;</code>	The cast operator is applied to the expression <code>3.7</code> . The operator returns the value <code>3</code> , which is assigned to the variable <code>x</code> .
	The cast operator is applied to the expression

```
number = (int)72.567;
```

72.567. The operator returns 72, which is used to initialize the variable number.

```
value = (float)x;
```

The cast operator returns the value in x, converted to a float. The converted value is assigned to the variable value.

```
value = (byte)number;
```

The cast operator returns the value in number, converted to a byte. The converted value is assigned to the variable value.

Note that when a cast operator is applied to a variable, it does not change the contents of the variable. It only returns the value stored in the variable, converted to the specified data type.

Recall from our earlier discussion that when both operands of a division are integers, the operation will result in integer division. This means that the result of the division will be an integer, with any fractional part of the result thrown away. For example, look at the following code:

```
int pies = 10, people = 4;  
double piesPerPerson;  
piesPerPerson = pies / people;
```

Although 10 divided by 4 is 2.5, this code will store 2 in the piesPerPerson variable. Because both pies and people are int variables, the result will be an int, and the fractional part will be thrown away. We can modify the code with a cast operator, however, so it gives the correct result as a floating-point value:

```
piesPerPerson = (double)pies / people;
```

The variable pies is an int and holds the value 10. The expression (double)pies returns the value in pies converted to a double. This means that one of the division operator's operands is a double, so the result of the division will be a double. The statement could also have been written as follows:

```
piesPerPerson = pies / (double)people;
```

In this statement, the cast operator returns the value of the `people` variable converted to a `double`. In either statement, the result of the division is a `double`.



Warning!

The cast operator can be applied to an entire expression enclosed in parentheses. For example, look at the following statement:

```
piesPerPerson = (double)(pies / people);
```

This statement does not convert the value in `pies` or `people` to a `double`, but converts the result of the expression `pies / people`. If this statement were used, an integer division operation would still have been performed. Here's why: The result of the expression `pies / people` is 2 (because integer division takes place). The value 2 converted to a `double` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `double`.

Mixed Integer Operations

One of the nuances of the Java language is the way it internally handles arithmetic operations on `int`, `byte`, and `short` variables. When values of the `byte` or `short` data types are used in arithmetic expressions, they are temporarily converted to `int` values. The result of an arithmetic operation using only a mixture of `byte`, `short`, or `int` values will always be an `int`.

For example, assume that `b` and `c` in the following expression are `short` variables:

```
b + c
```

Although both `b` and `c` are `short` variables, the result of the expression `b + c` is an `int`. This means that when the result of such an expression is stored in a variable, the variable must be an `int` or higher data type. For example, look

at the following code:

```
short firstNumber = 10,  
      secondNumber = 20,  
      thirdNumber;  
  
// The following statement causes an error!  
thirdNumber = firstNumber + secondNumber;
```

When this code is compiled, the following statement causes an error:

```
thirdNumber = firstNumber + secondNumber;
```

The error results from the fact that `thirdNumber` is a `short`. Although `firstNumber` and `secondNumber` are also `short` variables, the expression `firstNumber + secondNumber` results in an `int` value. The program can be corrected if `thirdNumber` is declared as an `int`, or if a cast operator is used in the assignment statement, as shown here:

```
thirdNumber = (short)(firstNumber + secondNumber);
```

Other Mixed Mathematical Expressions

In situations where a mathematical expression has one or more values of the `double`, `float`, or `long` data types, Java strives to convert all of the operands in the expression to the same data type. Let's look at the specific rules that govern evaluation of these types of expressions.

1. If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`. The result of the expression will be a `double`. For example, in the following statement, assume that `b` is a `double` and `c` is an `int`:

```
a = b + c;
```

The value in `c` will be converted to a `double` prior to the addition. The result of the addition will be a `double`, so the variable `a` must also be a

`double.`

2. If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`. The result of the expression will be a `float`. For example, in the following statement assume that `x` is a `short` and `y` is a `float`:

```
z = x * y;
```

The value in `x` will be converted to a `float` prior to the multiplication. The result of the multiplication will be a `float`, so the variable `z` must also be either a `double` or a `float`.

3. If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`. The result of the expression will be a `long`. For example, in the following statement assume that `a` is a `long` and `b` is a `short`:

```
c = a - b;
```

The variable `b` will be converted to a `long` prior to the subtraction. The result of the subtraction will be a `long`, so the variable `c` must also be a `long`, `float`, or `double`.

Checkpoint

1. 2.25 The following declaration appears in a program:

```
short totalPay, basePay = 500, bonus = 1000;
```

The following statement appears in the same program:

```
totalPay = basePay + bonus;
```

1. Will the statement compile properly or cause an error?
2. If the statement causes an error, why? How can you fix it?

2. 2.26 The variable `a` is a `float` and the variable `b` is a `double`. Write a statement that will assign the value of `b` to `a` without causing an error when the program is compiled.

2.8 Creating Named Constants with `final`

Concept:

The `final` key word can be used in a variable declaration to make the variable a named constant. Named constants are initialized with a value, and that value cannot change during the execution of the program.

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations, there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 8.2 percent? The programmer would have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A *named constant* is a variable whose value is read-only and cannot be changed during the program's execution. You can create such a variable in Java by using the `final` key word in the variable declaration. The word `final` is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word `final` appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so named constants are easily distinguishable from regular variable names.

When you declare a variable as `final`, you must either initialize it or assign it a value before you can use it. For example, the following code declares and initializes a `final` variable named `PRICE` before it is used in an output operation:

```
final double PRICE = 19.99;  
System.out.println("The price is " + PRICE);
```

The following code declares a `final` variable named `PRICE`, assigns it a value in a separate statement, then uses it in an output operation:

```
final double PRICE;  
PRICE = 19.99;  
System.out.println("The price is " + PRICE);
```

The following code will cause an error when it is compiled, however, because you cannot use a `final` variable before you have given it a value:

```
final double PRICE;  
System.out.println("The price is " + PRICE); // Error!  
PRICE = 19.99;
```

Once you have given a `final` variable a value, no statement in the program can change its value. The following code will cause a compiler error:

```
final double PRICE;  
PRICE = 19.99;  
System.out.println("The price is " + PRICE);  
PRICE = 12.99; // Error! Cannot change PRICE.
```

An advantage of using named constants is that they make programs more self-documenting. The following statement:

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that `balance` is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 8.2 percent, the declaration can be changed to the following:

```
final double INTEREST_RATE = 0.082;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will use the new value.

The `Math.PI` Named Constant

The `Math` class, which is part of the Java API, provides a predefined named constant, `Math.PI`. This constant is assigned the value 3.14159265358979323846, which is an approximation of the mathematical value pi. For example, look at the following statement:

```
area = Math.PI * radius * radius;
```

Assuming the `radius` variable holds the radius of a circle, this statement uses the `Math.PI` constant to calculate the area of the circle.

For more information about the `Math` class, see Appendix F, available on this book's online resource page at www.pearsonhighered.com/gaddis.

2.9 The String Class

Concept:

The `String` class allows you to create objects for holding strings. It also has various methods that allow you to work with strings.

You have already encountered strings and examined programs that display them on the screen, but let's take a moment to make sure you understand what a string is. A string is a sequence of characters. It can be used to represent any type of data that contains text, such as names, addresses, warning messages, and so forth. String literals are enclosed in double-quotation marks, such as the following:

```
"Hello World"  
"Joe Mahoney"
```

Although programs commonly encounter strings and must perform a variety of tasks with them, Java does not have a primitive data type for storing them in memory. Instead, the Java API provides a class for handling strings. You use this class to create objects that are capable of storing strings and performing operations on them. Before discussing this class, let's briefly discuss how classes and objects are related.

Objects Are Created from Classes

[Chapter 1](#) introduced you to objects as software entities that can contain attributes and methods. An object's attributes are data values that are stored in the object. An object's methods are procedures that perform operations on the object's attributes. Before an object can be created, however, it must be designed by a programmer. The programmer determines the attributes and

methods that are necessary, then creates a class that describes the object.

You have already seen classes used as containers for applications. A class can also be used to specify the attributes and methods that a particular type of object may have. Think of a class as a “blueprint” from which objects may be created. So a class is not an object, but a description of an object. When the program is running, it can use the class to create, in memory, as many objects as needed. Each object that is created from a class is called an *instance* of the class.

Tip:

Don’t worry if these concepts seem a little fuzzy to you. As you progress through this book, the concepts of classes and objects will be reinforced again and again.

The **String** Class

The class that is provided by the Java API for handling strings is named **String**. The first step in using the **String** class is to declare a variable of the **String** class data type. Here is an example of a **String** variable declaration:

```
String name;
```

Tip:

The **S** in **String** is written in uppercase letters. By convention, the first character of a class name is always written in uppercase letters.

This statement declares **name** as a **String** variable. Remember that **String** is a class, not a primitive data type. Let’s briefly look at the difference between primitive-type variables and class-type variables.

Primitive-Type Variables and Class-Type Variables

A variable of any type can be associated with an item of data. *Primitive-type variables* hold the actual data items with which they are associated. For example, assume that `number` is an `int` variable. The following statement stores the value 25 in the variable:

```
number = 25;
```

This is illustrated in [Figure 2-7](#).

Figure 2-7 A primitive-type variable holds the data with which it is associated

The `number` variable holds the actual data with which it is associated.

25

[Figure 2-7 Full Alternative Text](#)

A *class-type variable* does not hold the actual data item it is associated with, but holds the memory address of the data item it is associated with. If `name` is a `String` class variable, then `name` can hold the memory address of a `String` object. This is illustrated in [Figure 2-8](#).

Figure 2-8 A String class variable can hold the address of

a String object



[Figure 2-8 Full Alternative Text](#)

When a class-type variable holds the address of an object, it is said that the variable references the object. For this reason, class-type variables are commonly known as *reference variables*.

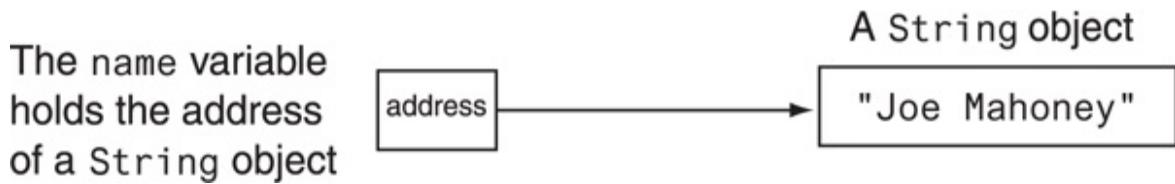
Creating a String Object

Anytime you write a string literal in your program, Java will create a `String` object in memory to hold it. You can create a `String` object in memory and store its address in a `String` variable with a simple assignment statement. Here is an example:

```
name = "Joe Mahoney";
```

Here, the string literal causes a `String` object to be created in memory with the value “Joe Mahoney” stored in it. Then, the assignment operator stores the address of that object in the `name` variable. After this statement executes, it is said that the `name` variable references a `String` object. This is illustrated in [Figure 2-9](#).

Figure 2-9 The name variable holds the address of a String object



[Figure 2-9 Full Alternative Text](#)

You can also use the = operator to initialize a String variable, as shown here:

```
String name = "Joe Mahoney";
```

This statement declares name as a String variable, creates a String object with the value “Joe Mahoney” stored in it, and assigns the object’s memory address to the name variable. [Code Listing 2-20](#) shows String variables being declared, initialized, and used in a `println` statement.

Code Listing 2-20 (StringDemo.java)

```

1 // A simple program demonstrating String objects.
2
3 public class StringDemo
4 {
5     public static void main(String[] args)
6     {
7         String greeting = "Good morning ";
8         String name = "Herman";
9
10        System.out.println(greeting + name);
11    }
12 }
```

Program Output

Good morning Herman

Because the String type is a class instead of a primitive data type, it provides numerous methods for working with strings. For example, the String class

has a method named `length` that returns the length of the string stored in an object. Assuming the `name` variable references a `String` object, the following statement stores the length of its string in the variable `stringSize` (assume that `stringSize` is an `int` variable):

```
stringSize = name.length();
```

This statement calls the `length` method of the object that `name` refers to. To *call* a method means to execute it. The general form of a method call is as follows:

```
referenceVariable.method(arguments...)
```

`referenceVariable` is the name of a variable that references an object, `method` is the name of a method, and `arguments...` is zero or more arguments that are passed to the method. If no arguments are passed to the method, as is the case with the `length` method, a set of empty parentheses must follow the name of the method.

The `String` class's `length` method *returns* an `int` value. This means that the method sends an `int` value back to the statement that called it. This value can be stored in a variable, displayed on the screen, or used in calculations. [Code Listing 2-21](#) demonstrates the `length` method.

Code Listing 2-21 (`StringLength.java`)

```
1 // This program demonstrates the String class's length method.  
2  
3 public class StringLength  
4 {  
5     public static void main(String[] args)  
6     {  
7         String name = "Herman";  
8         int stringSize;  
9  
10        stringSize = name.length();  
11        System.out.println(name + " has " + stringSize +  
12                            "characters.");
```

```
13 }  
14 }
```

Program Output

Herman has 6 characters.



Note:

The `String` class's `length` method returns the number of characters in the string, including spaces.

You will study the `String` class methods in detail in [Chapter 10](#), but let's look at a few more examples now. In addition to `length`, [Table 2-15](#) describes the `charAt`, `toLowerCase`, and `toUpperCase` methods.

Table 2-15 A few String class methods

Method	Description and Example
<code>charAt(index)</code>	The argument <code>index</code> is an <code>int</code> value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type <code>char</code> .
<code>charAt(index) Example:</code>	<pre>char letter; String name = "Herman"; letter = name.charAt(3);</pre>

```
char letter;  
String name = "Herman";  
letter = name.charAt(3);
```

After this code executes, the variable `letter` will hold the character '`m`'.

This method returns the number of characters in the string. The return value is of the type `int`.

Example:

```
length()    int stringSize;  
           String name = "Herman";  
           stringSize = name.length();
```

After this code executes, the `stringSize` variable will hold the value 6.

This method returns a new string that is the lowercase equivalent of the string contained in the calling object.

Example:

```
toLowerCase()  String bigName = "HERMAN";  
               String littleName = bigName.toLowerCase();
```

After this code executes, the object referenced by `littleName` will hold the string “herman”.

This method returns a new string that is the uppercase equivalent of the string contained in the calling object.

Example:

```
toUpperCase()  String littleName = "herman";  
               String bigName = littleName.toUpperCase();
```

After this code executes, the object referenced by

`bigName` will hold the string “HERMAN”.

The program in [Code Listing 2-22](#) demonstrates these methods.

Code Listing 2-22 (`StringMethods.java`)

```
1 // This program demonstrates a few of the String methods.  
2  
3 public class StringMethods  
4 {  
5     public static void main(String[] args)  
6     {  
7         String message = "Java is Great Fun!";  
8         String upper = message.toUpperCase();  
9         String lower = message.toLowerCase();  
10        char letter = message.charAt(2);  
11        int stringSize = message.length();  
12  
13        System.out.println(message);  
14        System.out.println(upper);  
15        System.out.println(lower);  
16        System.out.println(letter);  
17        System.out.println(stringSize);  
18    }  
19 }
```

Program Output

```
Java is Great Fun!  
JAVA IS GREAT FUN!  
java is great fun!  
v  
18
```



Checkpoint

1. 2.27 Write a statement that declares a `String` variable named `city`. The variable should be initialized so that it references an object with the string “San Francisco”.
2. 2.28 Assume that `stringLength` is an `int` variable. Write a statement that stores the length of the string referenced by the `city` variable (declared in [Checkpoint 2.27](#)) in `stringLength`.
3. 2.29 Assume that `oneChar` is a `char` variable. Write a statement that stores the first character in the string referenced by the `city` variable (declared in [Checkpoint 2.27](#)) in `oneChar`.
4. 2.30 Assume that `upperCity` is a `String` reference variable. Write a statement that stores the uppercase equivalent of the string referenced by the `city` variable (declared in [Checkpoint 2.27](#)) in `upperCity`.
5. 2.31 Assume that `lowerCity` is a `String` reference variable. Write a statement that stores the lowercase equivalent of the string referenced by the `city` variable (declared in [Checkpoint 2.27](#)) in `lowerCity`.

2.10 Scope

Concept:

A variable's scope is the part of the program that has access to the variable.

Every variable has a *scope*. The scope of a variable is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable's scope. The rules that define a variable's scope are complex, and you are only introduced to the concept here. In other chapters of the book, we will revisit this topic and expand on it.

So far, you have only seen variables declared inside the `main` method. Variables that are declared inside a method are called *local variables*. Later, you will learn about variables that are declared outside a method, but for now, let's focus on the use of local variables.

A local variable's scope begins at the variable's declaration, and ends at the end of the method in which the variable is declared. The variable cannot be accessed by statements that are outside this region. This means that a local variable cannot be accessed by code that is outside the method, or inside the method but before the variable's declaration. The program in [Code Listing 2-23](#) shows an example.

Code Listing 2-23 (Scope.java)

```
1 // This program can't find its variable.  
2  
3 public class Scope  
4 {  
5     public static void main(String[] args)  
6     {
```

```
7     System.out.println(value); // ERROR!
8     int value = 100;
9 }
10 }
```

The program does not compile because it attempts to send the contents of the variable `value` to `println` before the variable is declared. It is important to remember that the compiler reads your program from top to bottom. If it encounters a statement that uses a variable before the variable is declared, an error will result. To correct the program, the variable declaration must be written before any statement that uses it.



Note:

If you compile this program, the compiler will display an error message such as “cannot resolve symbol.” This means that the compiler has encountered a name for which it cannot determine a meaning.

Another rule that you must remember about local variables is that you cannot have two local variables with the same name in the same scope. For example, look at the following method.

```
public static void main(String[] args)
{
    // Declare a variable named number and display its value.
    int number = 7;
    System.out.println(number);
    // Declare another variable named number and display its value.
    int number = 100;           // ERROR!!!
    System.out.println(number); // ERROR!!!
}
```

This method declares a variable named `number` and initializes it with the value 7. The variable’s scope begins at the declaration statement and extends to the end of the method. Inside the variable’s scope a statement appears that declares another variable named `number`. This statement will cause an error because you cannot have two local variables with the same name in the same scope.

2.11 Comments

Concept:

Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

Comments are short notes that are placed in different parts of a program, explaining how those parts of the program work. Comments are not intended for the compiler. They are intended for programmers to read to help them understand the code. The compiler skips all the comments that appear in a program.

As a beginning programmer, you might resist the idea of writing a lot of comments in your programs. After all, it's a lot more fun to write code that actually does something! However, it is crucial that you take the extra time to write comments. They will almost certainly save you time in the future when you have to modify or debug the program. Proper comments make even large and complex programs easy to read and understand.

In Java, there are three types of comments: single-line comments, multiline comments, and documentation comments. Let's briefly discuss each one.

Single-Line Comments

You have already seen the first way to write comments in a Java program. You simply place two forward slashes (//) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. [Code Listing 2-24](#) shows that comments may be placed liberally throughout a program.

Code Listing 2-24 (Comment1.java)

```
1 // PROGRAM: Comment1.java
2 // Written by Herbert Dorfmann
3 // This program calculates company payroll
4
5 public class Comment1
6 {
7     public static void main(String[] args)
8     {
9         double payRate;      // Holds the hourly pay rate
10        double hours;       // Holds the hours worked
11        int employeeNumber; // Holds the employee number
12
13        // The remainder of this program is omitted.
14    }
15 }
```

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

Multiline Comments

The second type of comment in Java is the multiline comment. *Multiline comments* start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. [Code Listing 2-25](#) illustrates how multiline comments may be used.

Code Listing 2-25 (Comment2.java)

```
1 /*
2  PROGRAM: Comment2.java
3  Written by Herbert Dorfmann
4  This program calculates company payroll
5 */
6
```

```

7 public class Comment2
8 {
9     public static void main(String[] args)
10    {
11        double payRate; // Holds the hourly pay rate
12        double hours; // Holds the hours worked
13        int employeeNumber; // Holds the employee number
14
15        // The remainder of this program is omitted.
16    }
17 }

```

Unlike a comment started with `//`, a multiline comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to mark every line. Consequently, the multiline comment is inconvenient for writing single-line comments because you must type both a beginning and ending comment symbol.

Remember the following advice when using multiline comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Many programmers use asterisks or other characters to draw borders or boxes around their comments. This helps to visually separate the comments from the surrounding code. These are called block comments. [Table 2-16](#) shows four examples of block comments.

Table 2-16 Block comments

<code>/**</code>	<code>*****</code>
<code>* This program demonstrates the</code>	<code>// This program demonst</code>
<code>* way to write comments.</code>	<code>// way to write comment</code>
<code>*/</code>	<code>*****</code>

<code>-----</code>	<code>-----</code>
<code>// This program demonstrates the</code>	<code>// This program demonst</code>
<code>// way to write comments.</code>	<code>// way to write comment</code>

Documentation Comments

The third type of comment is known as a documentation comment.

Documentation comments can be read and processed by a program named javadoc, which comes with the JDK. The purpose of the javadoc program is to read Java source code files and generate attractively formatted HTML files that document the source code. If the source code files contain any documentation comments, the information in the comments becomes part of the HTML documentation, which may be viewed in a Web browser.

Any comment that starts with `/**` and ends with `*/` is considered a documentation comment. Normally, you write a documentation comment just before a class header, giving a brief description of the class. You also write a documentation comment just before each method header, giving a brief description of the method. For example, [Code Listing 2-26](#) shows a program with documentation comments. This program has a documentation comment just before the class header, and just before the `main` method header.

Code Listing 2-26 (Comment3.java)

```
1 /**
2  * This class creates a program that calculates company payroll
3 */
4
5 public class Comment3
6 {
7     /**
8      * The main method is the program's starting point.
9     */
10
11    public static void main(String[] args)
12    {
13        double payRate;    // Holds the hourly pay rate
14        double hours;     // Holds the hours worked
15        int employeeNumber; // Holds the employee number
```

```
16
17     // The Remainder of This Program is Omitted.
18 }
19 }
```

You run the javadoc program from the operating system command prompt. Here is the general format of the javadoc command:

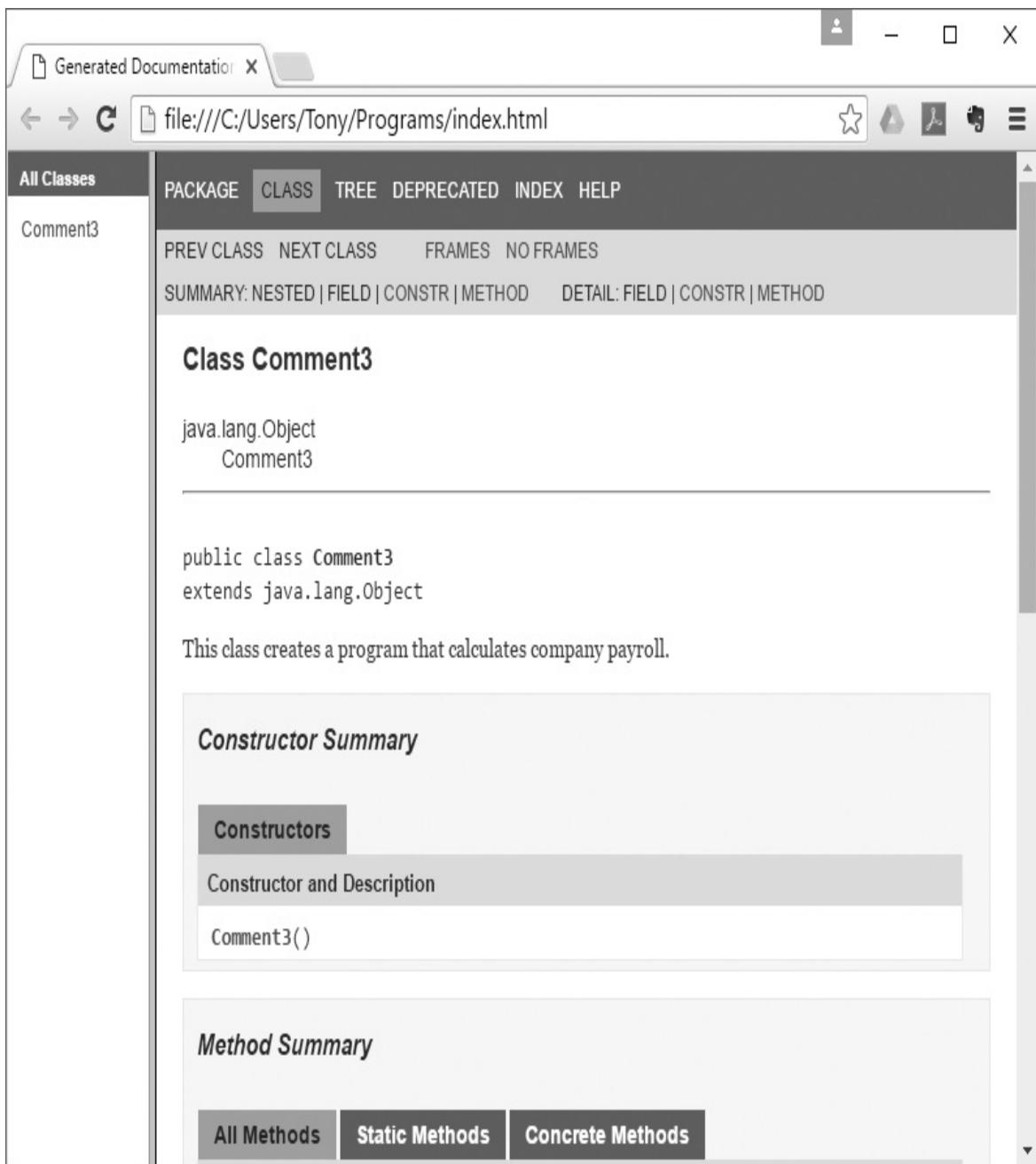
```
javadoc SourceFile.java
```

SourceFile.java is the name of a Java source code file, including the .java extension. The file will be read by javadoc, and documentation will be produced for it. For example, the following command will produce documentation for the *Comment3.java* source code file, which is shown in [Code Listing 2-26](#):

```
javadoc Comment3.java
```

After this command executes, several documentation files will be created in the same directory as the source code file. One of these files will be named *index.html*. [Figure 2-10](#) shows the *index.html* file being viewed in a Web browser. Notice that the text written in the documentation comments appears in the file.

Figure 2-10 Documentation generated by javadoc



[Figure 2-10 Full Alternative Text](#)

If you look at the JDK documentation, which are HTML files that you view in a Web browser, you will see that they are formatted in the same way as the files generated by javadoc. A benefit of using javadoc to document your source code is that your documentation will have the same professional look and feel as the standard Java documentation.

You can learn more about documentation comments and the javadoc utility by reading Appendix E, available on this book's online resource page at www.pearsonhighered.com/gaddis. From this point forward in the book, we will use simple block style comments and single-line comments in the example source code.



Checkpoint

1. 2.32 How do you write a single-line comment? How do you write a multiline comment? How do you write a documentation comment?
2. 2.33 How are documentation comments different from other types of comments?

2.12 Programming Style

Concept:

Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code.

In [Chapter 1](#), you learned that syntax rules govern the way a language may be used. The syntax rules of Java dictate how and where to place key words, semicolons, commas, braces, and other elements of the language. The compiler checks for syntax errors and, if there are none, generates byte code.

When the compiler reads a program, it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that spaces separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider [Code Listing 2-27](#) for example.

Code Listing 2-27 (Compact.java)

```
1 public class Compact {public static void main(String [] args){  
2     shares=220; double averagePrice=14.67; System.out.println(  
3     "There were "+shares+" shares sold at $" +averagePrice+  
4     " per share.");}}
```

Program Output

There were 220 shares sold at \$14.67 per share.

Although the program is syntactically correct (it doesn't violate any rules of

Java), it is very difficult to read. The same program is shown in [Code Listing 2-28](#), written in a more understandable style.

Code Listing 2-28 (Readable.java)

```
1 // This example is much more readable than Compact.java.  
2  
3 public class Readable  
4 {  
5     public static void main(String[] args)  
6     {  
7         int shares = 220;  
8         double averagePrice = 14.67;  
9  
10        System.out.println("There were " + shares +  
11                      " shares sold at $" +  
12                      averagePrice + " per share.");  
13    }  
14 }
```

Program Output

There were 220 shares sold at \$14.67 per share.

The term *programming style* usually refers to the way source code is visually arranged. It includes techniques for consistently putting spaces and indentations in a program so that visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in [Code Listing 2-28](#) that inside the class's braces each line is indented, and inside the `main` method's braces, each line is indented again. It is a common programming style to indent all the lines inside a set of braces, as shown in [Figure 2-11](#).

Figure 2-11 Indentation

```

// This example is much more readable than Compact.java.

public class Readable
{
    public static void main(String[] args)
    {
        int shares = 220;
        double averagePrice = 14.67;

        System.out.println("There were " + shares +
            " shares sold at $" +
            averagePrice + " per share.");
    }
}

```

[Figure 2-11 Full Alternative Text](#)

Another aspect of programming style is how to handle statements that are too long to fit on one line. Notice that the `println` statement is spread out over three lines. Extra spaces are inserted at the beginning of the statement's second and third lines, which indicate that they are continuations.

When declaring multiple variables of the same type with a single statement, it is a common practice to write each variable name on a separate line with a comment explaining the variable's purpose. Here is an example:

```

int          // To hold the Fahrenheit
fahrenheit, temperature
                  // To hold the Celsius
celsius,         temperature
                // To hold the Kelvin
kelvin;         temperature

```

You may have noticed in the example programs that a blank line is inserted between the variable declarations and the statements that follow them. This is intended to separate the declarations visually from the executable statements.

There are many other issues related to programming style. They will be presented throughout the book.

2.13 Reading Keyboard Input

Concept:

Objects of the Scanner class can be used to read input from the keyboard.

Previously, we discussed the `System.out` object and how it refers to the standard output device. The Java API has another object, `System.in`, which refers to the standard input device. The *standard input device* is normally the keyboard. You can use the `System.in` object to read keystrokes that have been typed at the keyboard. However, using `System.in` is not as simple and straightforward as using `System.out`, because the `System.in` object reads input only as byte values. This isn't very useful because programs normally require values of other data types as input. To work around this, you can use the `System.in` object in conjunction with an object of the `Scanner` class. The `Scanner` class is designed to read input from a source (such as `System.in`), and it provides methods that you can use to retrieve the input formatted as primitive values or strings.

First, you create a `Scanner` object and connect it to the `System.in` object. Here is an example of a statement that does just that:

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

Let's dissect the statement into two parts. The first part of the statement

```
Scanner keyboard
```

declares a variable named `keyboard`. The data type of the variable is `Scanner`. Because `Scanner` is a class, the `keyboard` variable is a class-type variable. Recall from our discussion on `String` objects that a class-type variable holds the memory address of an object. Therefore, the `keyboard` variable will be used to hold the address of a `Scanner` object. The second part of the

statement is as follows:

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

The first thing we see in this part of the statement is the assignment operator (=). The assignment operator will assign something to the keyboard variable. After the assignment operator we see the word new, which is a Java key word. The purpose of the new key word is to create an object in memory. The type of object that will be created is listed next. In this case, we see Scanner(System.in) listed after the new key word. This specifies that a Scanner object should be created, and it should be connected to the System.in object. The memory address of the object is assigned (by the = operator) to the variable keyboard. After the statement executes, the keyboard variable will reference the Scanner object that was created in memory.

[Figure 2-12](#) points out the purpose of each part of this statement. [Figure 2-13](#) illustrates how the keyboard variable references an object of the Scanner class.

Figure 2-12 The parts of the statement

This declares a variable named keyboard. The variable can reference an object of the Scanner class.

This creates a Scanner object in memory. The object will read input from System.in.

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

The = operator assigns the address of the Scanner object to the keyboard variable.

[Figure 2-12 Full Alternative Text](#)

Figure 2-13 The keyboard variable references a Scanner object



[Figure 2-13 Full Alternative Text](#)



Note:

In the preceding code, we chose `keyboard` as the variable name. There is nothing special about the name `keyboard`. We simply chose that name because we will use the variable to read input from the keyboard.

The `Scanner` class has methods for reading strings, bytes, integers, long integers, short integers, floats, and doubles. For example, the following code uses an object of the `Scanner` class to read an `int` value from the keyboard and assign the value to the `number` variable:

```
int number;  
Scanner keyboard = new Scanner(System.in);  
System.out.print("Enter an integer value: ");  
number = keyboard.nextInt();
```

The last statement shown here calls the `Scanner` class's `nextInt` method. The `nextInt` method formats an input value as an `int`, then returns that value. Therefore, this statement formats the input that was entered at the keyboard

as an int, then returns it. The value is assigned to the number variable.

[Table 2-17](#) lists several of the Scanner class's methods and describes their use.

Table 2-17 Some of the Scanner class methods

Method	Example and Description
	Example Usage:
nextByte	<pre>byte x; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a byte value: "); x = keyboard.nextByte();</pre>
	Description: Returns input as a byte.
	Example Usage:
nextDouble	<pre>double number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a double value: "); number = keyboard.nextDouble();</pre>
	Description: Returns input as a double.
	Example Usage:
nextFloat	<pre>float number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a float value: "); number = keyboard.nextFloat();</pre>

Description: Returns input as a float.

Example Usage:

```
nextInt      int number;
              Scanner keyboard = new Scanner(System.in);
              System.out.print("Enter an integer value: ");
              number = keyboard.nextInt();
```

Description: Returns input as an int.

Example Usage:

```
nextLine     String name;
              Scanner keyboard = new Scanner(System.in);
              System.out.print("Enter your name: ");
              name = keyboard.nextLine();
```

Description: Returns input as a String.

Example Usage:

```
nextLong     long number;
              Scanner keyboard = new Scanner(System.in);
              System.out.print("Enter a long value: ");
              number = keyboard.nextLong();
```

Description: Returns input as a long.

Example Usage:

```
nextShort    short number;
              Scanner keyboard = new Scanner(System.in);
              System.out.print("Enter a short value: ");
```

```
number = keyboard.nextShort();
```

Description: Returns input as a short.

Using the import Statement

There is one last detail about the Scanner class that you must know before you will be ready to use it. The Scanner class is not automatically available to your Java programs. Any program that uses the Scanner class should have the following statement near the beginning of the file, before any class definition:

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

This statement tells the Java compiler where in the Java library to find the Scanner class, and makes it available to your program.

[Code Listing 2-29](#) shows the Scanner class being used to read a String, an int, and a double.

Code Listing 2-29 (Payroll.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the Scanner class.
5  */
6
7 public class Payroll
8 {
9     public static void main(String[] args)
10    {
11        String name;      // To hold a name
12        int hours;       // Hours worked
13        double payRate;  // Hourly pay rate
14        double grossPay; // Gross pay
15    }
```

```

16 // Create a Scanner object to read input.
17 Scanner keyboard = new Scanner(System.in);
18
19 // Get the user's name.
20 System.out.print("What is your name? ");
21 name = keyboard.nextLine();
22
23 // Get the number of hours worked this week.
24 System.out.print("How many hours did you work this week? ");
25 hours = keyboard.nextInt();
26
27 // Get the user's hourly pay rate.
28 System.out.print("What is your hourly pay rate? ");
29 payRate = keyboard.nextDouble();
30
31 // Calculate the gross pay.
32 grossPay = hours * payRate;
33
34 // Display the resulting information.
35 System.out.println("Hello " + name);
36 System.out.println("Your gross pay is $" + grossPay);
37 }
38 }
```

Program Output with Example Input Shown in Bold

What is your name? **Joe Mahoney** 

How many hours did you work this week? **40** 

What is your hourly pay rate? **20** 

Hello Joe Mahoney

Your gross pay is \$800.0



Note:

Notice that each Scanner class method that we used waits for the user to press the  key before it returns a value. When the  key is pressed, the cursor automatically moves to the next line for subsequent output operations.

Reading a Character

Sometimes you will want to read a single character from the keyboard. For example, your program might ask the user a yes/no question, and specify that he or she type Y for yes or N for no. The `Scanner` class does not have a method for reading a single character, however. The approach that we will use in this book for reading a character is to use the `Scanner` class's `nextLine` method to read a string from the keyboard, then use the `String` class's `charAt` method to extract the first character of the string. This will be the character the user entered at the keyboard. Here is an example:

```
String input; // To hold a line of input
char answer; // To hold a single character
// Create a Scanner object for keyboard input.
Scanner keyboard = new Scanner(System.in);
// Ask the user a question.
System.out.print("Are you having fun? (Y=yes, N=no) ");
input = keyboard.nextLine(); // Get a line of input.
answer = input.charAt(0); // Get the first character.
```

The `input` variable references a `String` object. The last statement in this code calls the `String` class's `charAt` method to retrieve the character at position 0, which is the first character in the string. After this statement executes, the `answer` variable will hold the character that the user typed at the keyboard.

Mixing Calls to `nextLine` with Calls to Other `Scanner` Methods

When you call one of the `Scanner` class's methods to read a primitive value, such as `nextInt` or `nextDouble`, then call the `nextLine` method to read a string, an annoying and hard-to-find problem can occur. For example, look at the program in [Code Listing 2-30](#).

Code Listing 2-30

(InputProblem.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program has a problem reading input.
5 */
6
7 public class InputProblem
8 {
9     public static void main(String[] args)
10    {
11        String name; // To hold the user's name
12        int age; // To hold the user's age
13        double income; // To hold the user's income
14
15        // Create a Scanner object to read input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the user's age.
19        System.out.print("What is your age? ");
20        age = keyboard.nextInt();
21
22        // Get the user's income
23        System.out.print("What is your annual income? ");
24        income = keyboard.nextDouble();
25
26        // Get the user's name.
27        System.out.print("What is your name? ");
28        name = keyboard.nextLine();
29
30        // Display the information back to the user.
31        System.out.println("Hello " + name + ". Your age is " +
32                           age + " and your income is $" +
33                           income);
34    }
35 }
```

Program Output with Example Input Shown in Bold

What is your age? **24** 

What is your annual income? **50000.00** 

What is your name? Hello. Your age is 24 and your income is \$5000

Notice in the example output that the program first allows the user to enter his or her age. The statement in line 20 reads an `int` from the keyboard and stores the value in the `age` variable. Next, the user enters his or her income. The statement in line 24 reads a `double` from the keyboard and stores the value in the `income` variable. Then the user is asked to enter his or her name, but it appears that the statement in line 28 is skipped. The name is never read from the keyboard. This happens because of a slight difference in behavior between the `nextLine` method and the other `Scanner` class methods.

When the user types keystrokes at the keyboard, those keystrokes are stored in an area of memory that is sometimes called the *keyboard buffer*. Pressing the `Enter` key causes a newline character to be stored in the keyboard buffer. In the example running of the program in [Code Listing 2-30](#), the user was asked to enter his or her age, and the statement in line 20 called the `nextInt` method to read an integer from the keyboard buffer. Notice that the user typed 24 then pressed the `Enter` key. The `nextInt` method read the value 24 from the keyboard buffer, then stopped when it encountered the newline character. So the value 24 was read from the keyboard buffer, but the newline character was not read. The newline character remained in the keyboard buffer.

Next, the user was asked to enter his or her annual income. The user typed 50000.00 then pressed the `Enter` key. When the `nextDouble` method in line 24 executed, it first encountered the newline character that was left behind by the `nextInt` method. This does not cause a problem because the `nextDouble` method is designed to skip any leading newline characters it encounters. The method skips over the initial newline, reads the value 50000.00 from the keyboard buffer, and stops reading when it encounters the next newline character. This newline character is then left in the keyboard buffer.

Next, the user is asked to enter his or her name. In line 28, the `nextLine` method is called. The `nextLine` method, however, is not designed to skip over an initial newline character. If a newline character is the first character that the `nextLine` method encounters, then nothing will be read. Because the `nextDouble` method (back in line 24) left a newline character in the keyboard buffer, the `nextLine` method will not read any input. Instead, it will immediately terminate, and the user will not be given a chance to enter his or

her name.

Although the details of this problem might seem confusing, the solution is easy. The program in [Code Listing 2-31](#) is a modification of [Code Listing 2-30](#), with the input problem fixed.

Code Listing 2-31 (**CorrectedInputProblem.java**)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program correctly read numeric and string input.
5  */
6
7 public class CorrectedInputProblem
8 {
9     public static void main(String[] args)
10    {
11        String name; // To hold the user's name
12        int age; // To hold the user's age
13        double income; // To hold the user's income
14
15        // Create a Scanner object to read input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the user's age.
19        System.out.print("What is your age? ");
20        age = keyboard.nextInt();
21
22        // Get the user's income
23        System.out.print("What is your annual income? ");
24        income = keyboard.nextDouble();
25
26        // Consume the remaining newline.
27        keyboard.nextLine();
28
29        // Get the user's name.
30        System.out.print("What is your name? ");
31        name = keyboard.nextLine();
32
33        // Display the information back to the user.
34        System.out.println("Hello " + name + ". Your age is " +
```

```
35         age + " and your income is $" +
36         income);
37     }
38 }
```

Program Output with Example Input Shown in Bold

What is your age? **24** 

What is your annual income? **50000.00** 

What is your name? **Mary Simpson** 

Hello Mary Simpson. Your age is 24 and your income is \$50000.0

Notice that after the user's income is read by the `nextDouble` method in line 24, the `nextLine` method is called in line 27. The purpose of this call is to consume, or remove, the newline character that remains in the keyboard buffer. Then, in line 31, the `nextLine` method is called again. This time it correctly reads the user's name.



Note:

Notice that in line 27, where we consume the remaining newline character, we do not assign the method's return value to any variable. This is because we are simply calling the method to remove the newline character, and we do not need to keep the method's return value.

2.14 Dialog Boxes

Concept:

The JOptionPane class allows you to quickly display a dialog box, which is a small graphical window displaying a message or requesting input.

A *dialog box* is a small graphical window that displays a message to the user or requests input. You can quickly display dialog boxes with the JOptionPane class. In this section, we will discuss the following types of dialog boxes and how you can display them using JOptionPane:

- Message Dialog A dialog box that displays a message; an OK button is also displayed
- Input Dialog A dialog box that prompts the user for input and provides a text field where input is typed; an OK button and a Cancel button are also displayed

[Figure 2-14](#) shows an example of each type of dialog box.

Figure 2-14 A message dialog and an input dialog



[Figure 2-14 Full Alternative Text](#)

The `JOptionPane` class is not automatically available to your Java programs. Any program that uses the `JOptionPane` class must have the following statement near the beginning of the file:

```
import javax.swing.JOptionPane;
```

This statement tells the compiler where to find the `JOptionPane` class, and makes it available to your program.

Displaying Message Dialogs

The `showMessageDialog` method is used to display a message dialog. Here is a statement that calls the method:

```
JOptionPane.showMessageDialog(null, "Hello World");
```

We will always pass the key word `null` as the first argument. This causes the dialog box to be displayed in the center of the screen. The second argument is the message that we wish to display in the dialog box. This code will cause the dialog box in [Figure 2-15](#) to appear. When the user clicks the `OK` button, the dialog box will close.

Figure 2-15 Message dialog



Displaying Input Dialogs

An input dialog is a quick and simple way to ask the user to enter data. You use the `JOptionPane` class's `showInputDialog` method to display an input dialog. The following code calls the method:

```
String name;  
name = JOptionPane.showInputDialog("Enter your name.");
```

The argument passed to the method is a message to display in the dialog box. This statement will cause the dialog box shown in [Figure 2-16](#) to be displayed in the center of the screen. If the user clicks the OK button, `name` will reference the string value entered by the user into the text field. If the user clicks the Cancel button, `name` will reference the special value `null`.

Figure 2-16 Input dialog



An Example Program

The program in [Code Listing 2-32](#) demonstrates how to use both types of dialog boxes. This program uses input dialogs to ask the user to enter his or her first and last names, then displays a greeting with a message dialog. When this program executes, the dialog boxes shown in [Figure 2-17](#) will be displayed, one at a time.

Figure 2-17 Dialog boxes displayed by the NamesDialog program

The first dialog box appears as shown here. The user types Joe then clicks OK.



The second dialog box appears, as shown here. In this example, the user types Clondike then clicks OK.



The third dialog box appears, as shown here, displaying a greeting.



[Figure 2-17 Full Alternative Text](#)

Code Listing 2-32 (NamesDialog.java)

```
1 import javax.swing.JOptionPane;
2
3 /**
4  * This program demonstrates using dialogs
5  * with JOptionPane.
6 */
7
8 public class NamesDialog
9 {
10    public static void main(String[] args)
11    {
12        String firstName, lastName;
13
14        // Get the user's first name.
15        firstName =
16            JOptionPane.showInputDialog("What is your first name?");
17
18        // Get the user's last name.
19        lastName =
20            JOptionPane.showInputDialog("What is your last name?");
21
22        // Display a greeting
23        JOptionPane.showMessageDialog(null, "Hello " + firstName +
24                                lastName);
25        System.exit(0);
26    }
27 }
```

Notice the last statement in the `main` method:

```
System.exit(0);
```

This statement causes the program to end, and is required if you use the `JOptionPane` class to display dialog boxes. Unlike a console program, a program that uses `JOptionPane` does not automatically stop executing when the end of the `main` method is reached because the `JOptionPane` class causes

an additional task to run in the JVM. If the `System.exit` method is not called, this task, also known as a *thread*, will continue to execute, even after the end of the `main` method has been reached.

The `System.exit` method requires an integer argument. This argument is an exit code that is passed back to the operating system. Although this code is usually ignored, it can be used outside the program to indicate whether the program ended successfully or as the result of a failure. The value 0 traditionally indicates that the program ended successfully.

Converting String Input to Numbers

Unlike the `Scanner` class, the `JOptionPane` class does not have different methods for reading values of different data types as input. The `showInputDialog` method always returns the user's input as a `String`, even if the user enters numeric data. For example, if the user enters the number 72 into an input dialog, the `showInputDialog` method will return the string "72". This can be a problem if you wish to use the user's input in a math operation because, as you know, you cannot perform math on strings. In such a case, you must convert the input to a numeric value. To convert a string value to a numeric value, you use one of the methods listed in [Table 2-18](#).

Table 2-18 Methods for converting strings to numbers

Method	Use This Method to ... Convert a string	Example Code
<code>Byte.parseByte</code>	<code>byte num;</code>	

	to a byte.
Double.parseDouble	Convert a string to a double. double.
Float.parseFloat	Convert a string to a float. float.
Integer.parseInt	Convert a string to an int. int.
Long.parseLong	Convert a string to a long. long.
Short.parseShort	Convert a string to a short. short.



Note:

The methods in [Table 2-18](#) are part of Java's wrapper classes, which you will learn more about in [Chapter 8](#).

Here is an example of how you would use the `Integer.parseInt` method to convert the value returned from the `JOptionPane.showInputDialog` method to an `int`:

```
int number;
String str;
str = JOptionPane.showInputDialog("Enter a number.");
```

```
number = Integer.parseInt(str);
```

After this code executes, the `number` variable will hold the value entered by the user, converted to an `int`. Here is an example of how you would use the `Double.parseDouble` method to convert the user's input to a `double`:

```
double price;
String str;
str = JOptionPane.showInputDialog("Enter the retail price.");
price = Double.parseDouble(str);
```

After this code executes, the `price` variable will hold the value entered by the user, converted to a `double`. [Code Listing 2-33](#) shows a complete program. This is a modification of the `Payroll.java` program in [Code Listing 2-29](#). When this program executes, the dialog boxes shown in [Figure 2-18](#) will be displayed, one at a time.

Figure 2-18 Dialog boxes displayed by PayrollDialog.java

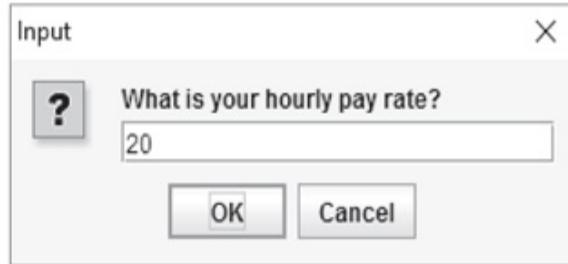
The first dialog box appears as shown here. The user enters his or her name then clicks OK.



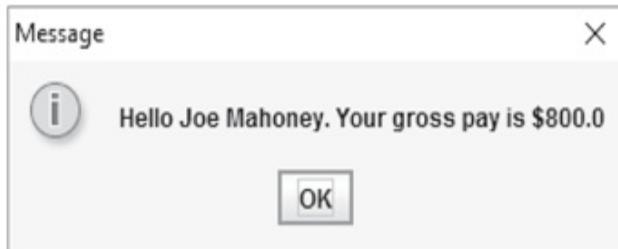
The second dialog box appears, as shown here. The user enters the number of hours worked then clicks OK.



The third dialog box appears, as shown here. The user enters his or her hourly pay rate then clicks OK.



The fourth dialog box appears, as shown here.



[Figure 2-18 Full Alternative Text](#)

Code Listing 2-33 (PayrollDialog.java)

```
1 import javax.swing.JOptionPane;
2
3 /**
4  * This program demonstrates using dialogs
5  * with JOptionPane.
6 */
```

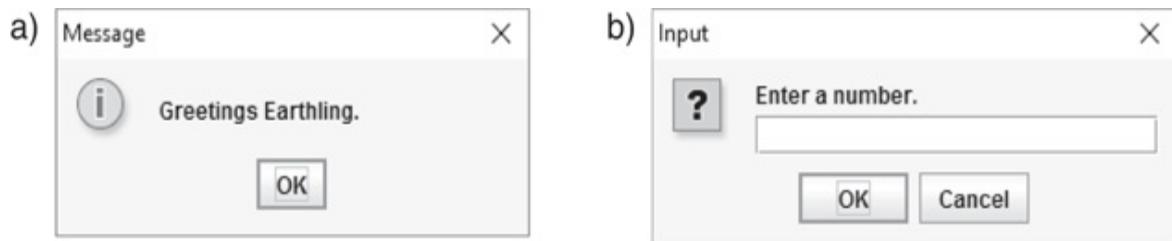
```
7
8 public class PayrollDialog
9 {
10 public static void main(String[] args)
11 {
12     String inputString;      // For reading input
13     String name;            // The user's name
14     int hours;              // The number of hours worked
15     double payRate;         // The user's hourly pay rate
16     double grossPay;        // The user's gross pay
17
18     // Get the user's name.
19     name = JOptionPane.showInputDialog("What is " +
20             "your name? ");
21
22     // Get the hours worked.
23     inputString =
24         JOptionPane.showInputDialog("How many hours " +
25             "did you work this week? ");
26
27     // Convert the input to an int.
28     hours = Integer.parseInt(inputString);
29
30     // Get the hourly pay rate.
31     inputString =
32         JOptionPane.showInputDialog("What is your " +
33             "hourly pay rate? ");
34
35     // Convert the input to a double.
36     payRate = Double.parseDouble(inputString);
37
38     // Calculate the gross pay.
39     grossPay = hours * payRate;
40
41     // Display the results.
42     JOptionPane.showMessageDialog(null, "Hello " +
43             name + ". Your gross pay is $" +
44             grossPay);
45
46     // End the program.
47     System.exit(0);
48 }
49 }
```



Checkpoint

1. 2.34 What is the purpose of the following types of dialog boxes?
 - Message dialog
 - Input dialog
2. 2.35 Write code that will display each of the dialog boxes shown in [Figure 2-19](#).
3. 2.36 Write code that displays an input dialog asking the user to enter his or her age. Convert the input value to an `int` and store it in an `int` variable named `age`.
4. 2.37 What `import` statement do you write in a program that uses the `JOptionPane` class?

Figure 2-19 Dialog boxes for [Checkpoint 2.35](#)



[Figure 2-19 Full Alternative Text](#)

2.15 Displaying Formatted Output with `System.out.printf` and `String.format`

Concept:

The `System.out.printf` method allows you to format output in a variety of ways. The `String.format` method allows you to format a string, without displaying it. The string can be displayed at a later time.

When you display numbers with the `System.out.println` or `System.out.print` methods, you have little control over the way the numbers appear. For example, a value of the `double` data type can be displayed with as many as 15 decimal places, as demonstrated by the following code:

```
double number = 10.0 / 6.0;  
System.out.println(number);
```

This code will display:

1.6666666666666667

Quite often, you want to format numbers so they are displayed in a particular way. For example, you might want to round a floating-point value number to a specific number of decimal places, or insert comma separators to make a number easier to read. Fortunately, Java gives you a way to do just that, and more, with the `System.out.printf` method. The method's general format is as follows:

`System.out.printf(FormatString, ArgumentList)`

In the general format, `FormatString` is a string that contains text and/or

special formatting specifiers. *ArgumentList* is a list of zero or more additional arguments, which will be formatted according to the format specifiers listed in the format string.

The simplest way you can use the `System.out.printf` method is with only a format string, and no additional arguments. Here is an example:

```
System.out.printf("I love Java programming.");
```

The format string in this example is "I love Java programming". This method call does not perform any special formatting; however, it simply prints the string "I love Java programming.". Using the method in this fashion is exactly like using the `System.out.print` method.

In most cases, you will call the `System.out.printf` method in the following manner:

- The format string will contain one or more format specifiers. A *format specifier* is a placeholder for a value that will be inserted into the string when it is displayed.
- After the format string, one or more additional arguments will appear. Each of the additional arguments will correspond to a format specifier that appears inside the format string.

The following code shows an example:

```
double sales = 12345.67;
System.out.printf("Our sales is %f for the day.\n", sales);
```

Notice the following characteristics of the `System.out.printf` method call:

- Inside the format string, the `%f` is a format specifier. The letter `f` indicates that a floating-point value will be inserted into the string when it is displayed.
- Following the format string, the `sales` variable is passed as an argument. This argument corresponds to the `%f` format specifier that appears inside the format string.

When the `System.out.printf` method executes, the `%f` will not be displayed on the screen. In its place, the value of the `sales` argument will be displayed. Here is the output of the code:

```
Our sales is 12345.670000 for the day.
```

The diagram in [Figure 2-20](#) shows how the `sales` variable corresponds to the `%f` format specifier.

Figure 2-20 The value of the sales variable is displayed in the place of the `%f` format specifier



```
System.out.printf ("Our sales are %f for the day.\n", sales);
```

Here is another example:

```
double temp1 = 72.5, temp2 = 83.7;  
System.out.printf("The temperatures are %f and %f degrees.\n", te
```

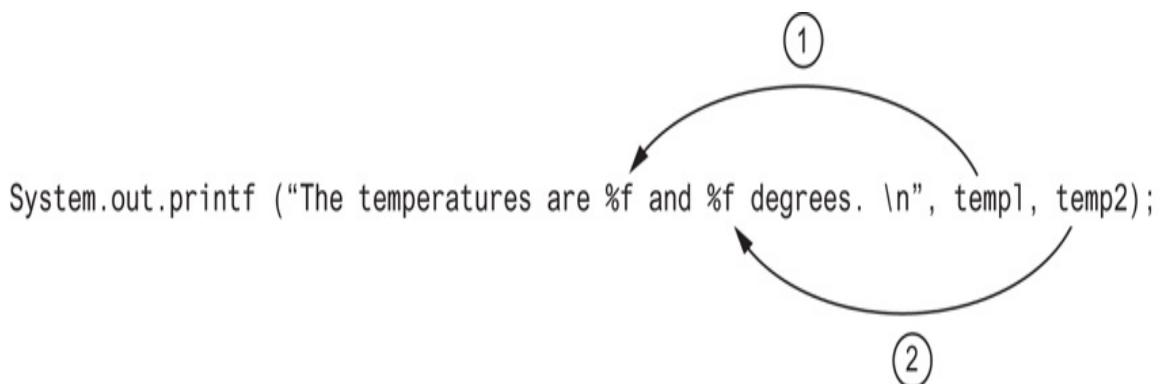
First, notice this example uses two `%f` format specifiers in the format string. Also, notice that the two additional arguments appear after the format string. The value of the first argument, `temp1`, will be printed in place of the first `%f`, and the value of the second argument, `temp2`, will be printed in place of the second `%f`. The code will produce the following output:

```
The temperatures are 72.500000 and 83.700000 degrees.
```

There is a one-to-one correspondence between the format specifiers and the arguments that appear after the format string. The diagram in [Figure 2-21](#)

shows how the first format specifier corresponds to the first argument after the format string (the `temp1` variable), and the second format specifier corresponds to the second argument after the format string (the `temp2` variable).

Figure 2-21 The format specifiers and their corresponding arguments



[Figure 2-21 Full Alternative Text](#)

The following code shows another example:

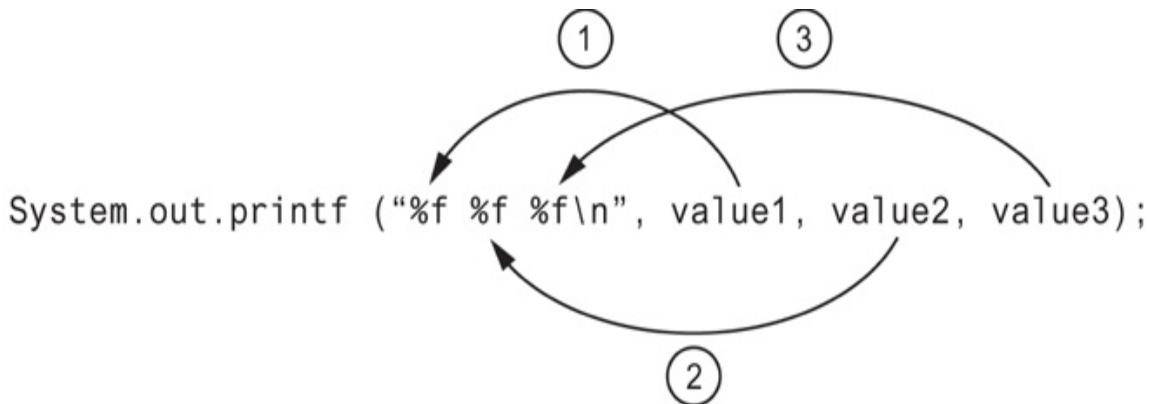
```
double value1 = 3.0;
double value2 = 6.0;
double value3 = 9.0;
System.out.printf("%f %f %f\n", value1, value2, value3);
```

In the `System.out.printf` method call, there are three format specifiers and three additional arguments after the format string. This code will produce the following output:

3.000000 6.000000 9.000000

The diagram in [Figure 2-22](#) shows how the format specifiers correspond to the arguments that appear after the format string.

Figure 2-22 The format specifiers and their corresponding arguments



[Figure 2-22 Full Alternative Text](#)

The previous examples demonstrated how to format floating-point numbers with the `%f` format specifier. The letter `f` in the format specifier is a *conversion character* that indicates the data type of the argument that is being formatted. You use the `f` conversion character with any argument that is a `float` or a `double`.

If you want to format an integer value, you must use the `%d` format specifier. The `d` conversion character stands for decimal integer, and it can be used with arguments of the `int`, `short`, and `long` data types. Here is an example that displays an `int`:

```
int hours = 40;  
System.out.printf("I worked %d hours this week.\n", hours);
```

In this example, the `%d` format specifier corresponds with the `hours` argument. This code will display the following:

I worked 40 hours this week.

Here is an example that displays two `int` values:

```
int dogs = 2;  
int cats = 4;  
System.out.printf("We have %d dogs and %d cats.\n", dogs, cats);
```

This code will display the following:

We have 2 dogs and 4 cats.

Keep in mind that %f must be used with floating-point values, and %d must be used with integer values. Otherwise, an error will occur at runtime.

Format Specifier Syntax

In the previous examples, you saw how format specifiers correspond to the arguments that appear after the format string. Now you can learn how to use format specifiers to actually format the values to which they correspond. When displaying numbers, the general syntax for writing a format specifier is:

%[flags][width].[precision]conversion

The items that appear inside brackets are optional. Here is a summary of each item:

- %—All format specifiers begin with a % character.
- *flags*—After the % character, one or more optional flags may appear. Flags cause the value to be formatted in a variety of ways.
- *width*—After any flags, you can optionally specify the minimum field width for the value.
- *.precision*—If the value is a floating-point number, after the minimum field width, you can optionally specify the precision. This is the number of decimal places to which the number should be rounded.
- *conversion*—All format specifiers must end with a conversion character, such as f for floating-point value, or d for decimal integer.

Let's take a closer look at each of the optional items, beginning with precision.

Precision

You probably noticed in the previous examples that the %f format specifier causes floating-point values to be displayed with six decimal places. You can change the number of decimal places that are displayed, as shown in the following example:

```
double temp = 78.42819;  
System.out.printf("The temperature is %.2f degrees.\n", temp);
```

Notice that this example doesn't use the regular %f format specifier, but uses %.2f instead. The .2 that appears between the % and the f specifies the *precision* of the displayed value. It will cause the value of the temp variable to be rounded to two decimal places. This code will produce the following output:

The temperature is 78.43 degrees.

The following example displays the same value, rounded to one decimal place:

```
double temp = 78.42819;  
System.out.printf("The temperature is %.1f degrees.\n", temp);
```

This code will produce the following output:

The temperature is 78.4 degrees.

The following code shows another example:

```
double value1 = 123.45678;  
double value2 = 123.45678;  
double value3 = 123.45678;  
System.out.printf("%.1f %.2f %.3f\n", value1, value2, value3);
```

In this example, value1 is rounded to one decimal place, value2 is rounded

to two decimal places, and `value3` is rounded to three decimal places. This code will produce the following output:

```
123.5 123.46 123.457
```

Keep in mind that you can specify precision only with floating-point values. If you specify a precision with the `%d` format specifier, an error will occur at runtime.

Specifying a Minimum Field Width

A format specifier can also include a *minimum field width*, which is the minimum number of spaces that should be used to display the value. The following example prints a floating-point number in a field that is 20 spaces wide:

```
double number = 12345.6789;  
System.out.printf("The number is:%20f\n", number);
```

Notice that the number 20 appears in the format specifier, between the % and the f. This code will produce the following output:

```
The number is:      12345.678900
```

In this example, the 20 that appears inside the `%f` format specifier indicates that the number should be displayed in a field that is a minimum of 20 spaces wide. This is illustrated in [Figure 2-23](#).

Figure 2-23 The number is displayed in a field that is 20 spaces wide

The number is:  12345.678900

The number is displayed in a field that is 20 spaces wide.

In this case, the number that is displayed is shorter than the field in which it is displayed. The number 12345.678900 uses only 12 spaces on the screen, but it is displayed in a field that is 20 spaces wide. When this is the case, the number will be right-justified in the field. If a value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it. The following example prints a floating-point number in a field that is only one space wide:

```
double number = 12345.6789;  
System.out.printf("The number is:%1f\n", number);
```

The value of the number variable requires more than one space, however, so the field width is expanded to accommodate the entire number. This code will produce the following output:

The number is:12345.678900

You can specify a minimum field width for integers, as well as floating-point values. The following example displays an integer with a minimum field width of six characters:

```
int number = 200;  
System.out.printf("The number is:%6d", number);
```

This code will display the following:

The number is: 200

Combining Minimum Field Width and Precision in the Same Format Specifier

When specifying the minimum field width and the precision of a floating-point number in the same format specifier, remember that the field width must appear first, followed by the precision. For example, the following code displays a number in a field of 12 spaces, rounded to two decimal places:

```
double number = 12345.6789;
System.out.printf("The number is:%12.2f\n", number);
```

This code will produce the following output:

```
The number is: 12345.68
```

Field widths can help when you need to print numbers aligned in columns. For example, look at [Code Listing 2-34](#). Each of the variables is displayed in a field that is eight spaces wide, and rounded to two decimal places. The numbers appear aligned in a column.

Code Listing 2-34 (Columns.java)

```
1 /**
2  * This program displays a variety of
3  * floating-point numbers in a column
4  * with their decimal points aligned.
5 */
6
7 public class Columns
8 {
9     public static void main(String[] args)
10    {
11        // Declare a variety of double variables.
12        double num1 = 127.899;
13        double num2 = 3465.148;
14        double num3 = 3.776;
15        double num4 = 264.821;
16        double num5 = 88.081;
17        double num6 = 1799.999;
18
19        // Display each variable in a field of
20        // 8 spaces with 2 decimal places.
21        System.out.printf("%8.2f\n", num1);
22        System.out.printf("%8.2f\n", num2);
23        System.out.printf("%8.2f\n", num3);
```

```
24     System.out.printf("%8.2f\n", num4);
25     System.out.printf("%8.2f\n", num5);
26     System.out.printf("%8.2f\n", num6);
27 }
28 }
```

Program Output

```
127.90
3465.15
 3.78
 264.82
 88.08
1800.00
```

Flags

There are several optional flags that you can insert into a format specifier to cause a value to be formatted in a particular way. In this book, we will use flags for the following purposes:

- To display numbers with comma separators
- To pad numbers with leading zeros
- To left justify numbers

If you use a flag in a format specifier, you must write the flag before the field width and the precision.

Comma Separators

Large numbers are easier to read if they are displayed with comma separators. You can format a number with comma separators by inserting a comma (,) flag into the format specifier. Here is an example:

```
double amount = 1234567.89;
```

```
System.out.printf("%,f\n", amount);
```

This code will produce the following output:

```
1,234,567.890000
```

Quite often, you will want to format a number with comma separators, and round the number to a specific number of decimal places. You can accomplish this by inserting a comma, followed by the precision value, into the %f format specifier, as shown in the following example:

```
double sales = 28756.89;
System.out.printf("Sales for the month are %,.2f\n", sales);
```

This code will produce the following output:

```
Sales for the month are 28,756.89
```

[Code Listing 2-35](#) demonstrates how the comma separator and a precision of two decimal places can be used to format a number as a currency amount.

Code Listing 2-35 (CurrencyFormat.java)

```
1 /**
2  * This program demonstrates how to use the System.out.printf
3  * method to format a number as currency.
4 */
5
6 public class CurrencyFormat
7 {
8     public static void main(String[] args)
9     {
10         double monthlyPay = 5000.0;
11         double annualPay = monthlyPay * 12;
12         System.out.printf("Your annual pay is $%,.2f\n", annualPay)
13     }
14 }
```

Program Output

Your annual pay is \$60,000.00

The following example displays a floating-point number with comma separators, in a field of 15 spaces, rounded to two decimal places:

```
double amount = 1234567.8901;  
System.out.printf("%,15.2f\n", amount);
```

This code will produce the following output:

1,234,567.89

The following example displays an int with a minimum field width of six characters:

```
int number = 200;  
System.out.printf("The number is:%6d", number);
```

This code will display the following:

The number is: 200

The following example displays an int with comma separators, with a minimum field width of 10 characters:

```
int number = 20000;  
System.out.printf("The number is:%,10d", number);
```

This code will display the following:

The number is: 20,000

Padding Numbers with Leading Zeros

Sometimes, when a number is shorter than the field in which it is displayed,

you want to pad the number with leading zeros. If you insert a 0 flag into a format specifier, the resulting number will be padded with leading zeros, if it is shorter than the field width. The following code shows an example:

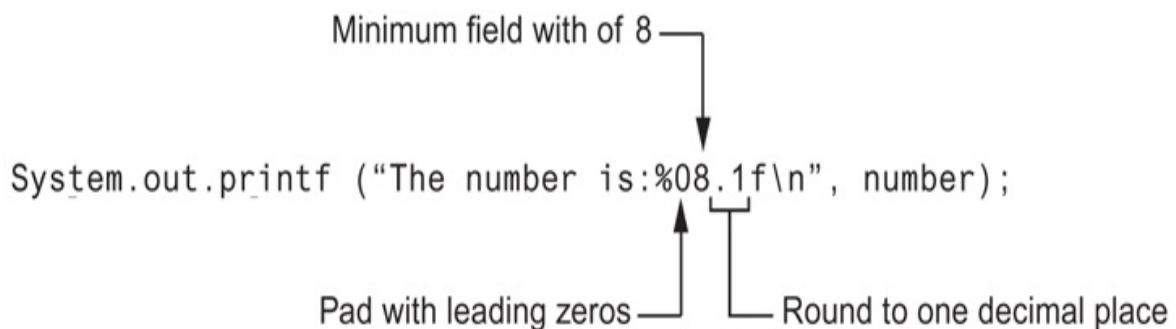
```
double number = 123.4;  
System.out.printf("The number is:%08.1f\n", number);
```

This code will produce the following output:

```
The number is:000123.4
```

The diagram in [Figure 2-24](#) shows the purpose of each part of the format specifier in the previous example.

Figure 2-24 Format specifier that pads with leading zeros



[Figure 2-24 Full Alternative Text](#)

The following example displays an int padded with leading zeros, with a minimum field width of seven characters:

```
int number = 1234;  
System.out.printf("The number is:%07d", number);
```

This code will display the following:

```
The number is:0001234
```

The program in [Code Listing 2-36](#) shows another example. This program displays a variety of floating-point numbers with leading zeros, in a field of nine spaces, rounded to two decimal places.

Code Listing 2-36 (LeadingZeros.java)

```
1 /**
2  * This program displays numbers padded with leading zeros.
3 */
4
5 public class LeadingZeros
6 {
7     public static void main(String[] args)
8     {
9         // Declare a variety of double variables.
10        double number1 = 1.234;
11        double number2 = 12.345;
12        double number3 = 123.456;
13
14        // Display each variable with leading
15        // zeros, in a field of 9 spaces, rounded
16        // to 2 decimal places.
17        System.out.printf("%09.2f\n", number1);
18        System.out.printf("%09.2f\n", number2);
19        System.out.printf("%09.2f\n", number3);
20    }
21 }
```

Program Output

```
000001.23
000012.35
000123.46
```

Left-Justified Numbers

By default, when a number is shorter than the field in which it is displayed,

the number is right-justified within that field. If you want a number to be left-justified within its field, you insert a minus sign (-) flag into the format specifier. [Code Listing 2-37](#) shows an example.

Code Listing 2-37 (LeftJustified.java)

```
1 /**
2  * This program displays a variety of
3  * numbers left-justified in columns.
4 */
5
6 public class LeftJustified
7 {
8     public static void main(String[] args)
9     {
10         // Declare a variety of int variables.
11         int num1 = 123;
12         int num2 = 12;
13         int num3 = 45678;
14         int num4 = 456;
15         int num5 = 1234567;
16         int num6 = 1234;
17
18         // Display each variable left-justified
19         // in a field of 8 spaces.
20         System.out.printf("%-8d%-8d\n", num1, num2);
21         System.out.printf("%-8d%-8d\n", num3, num4);
22         System.out.printf("%-8d%-8d\n", num5, num6);
23     }
24 }
```

Program Output

```
123 12
45678 456
1234567 1234
```

Formatting String Arguments

If you wish to print a string argument, use the %s format specifier. Here is an example:

```
String name = "Ringo";
System.out.printf("Your name is %s\n", name);
```

This code produces the following output:

```
Your name is Ringo
```

You can also use a field width when printing strings. For example, look at the following code:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%10s%10s\n", name1, name2);
System.out.printf("%10s%10s\n", name3, name4);
System.out.printf("%10s%10s\n", name5, name6);
```

The %10s format specifier prints a string in a field that is 10 spaces wide. This code displays the values of the variables in a table with three rows and two columns. Each column has a width of 10 spaces. Here is the output of the code:

George	Franklin
Jay	Ozzy
Carmine	Dee

Notice that the strings are right-justified. You can use the minus flag (-) to left-justify a string within its field, as the following code demonstrates:

```
String name1 = "George";
String name2 = "Franklin";
String name3 = "Jay";
String name4 = "Ozzy";
String name5 = "Carmine";
String name6 = "Dee";
System.out.printf("%-10s%-10s\n", name1, name2);
System.out.printf("%-10s%-10s\n", name3, name4);
```

```
System.out.printf("%-10s%-10s\n", name5, name6);
```

Here is the output of the code:

```
George Franklin  
Jay Ozzy  
Carmine Dee
```

The following example shows how you can print arguments of different data types:

```
int hours = 40;  
double pay = hours * 25;  
String name = "Jay";  
System.out.printf("Name: %s, Hours: %d, Pay: $%,.2f\n",  
    name, hours, pay);
```

In this example, we are displaying a `String`, an `int`, and a `double`. The code will produce the following output:

```
Name: Jay, Hours: 40, Pay: $1,000.00
```



Note:

The format specifiers we have shown in this section are the basic ones. Java provides much more powerful format specifiers for more complex formatting needs. The API documentation gives an overview of them all.

The `String.format` Method

The `System.out.printf` method formats a string and displays it in the console window. Sometimes you need to format a string without displaying it in the console. For example, you might need to display formatted output in a graphical interface, such as a message dialog. When this is the case, you can use the `String.format` method.

The `String.format` method works exactly like the `System.out.printf`

method, except that it does not display the formatted string on the screen. Instead, it returns a reference to the formatted string. You can assign the reference to a variable, and use it later. Here is the method's general format:

```
String.format(FormatString, ArgumentList)
```

In the general format, *FormatString* is a string that contains text, special formatting specifiers, or both. *ArgumentList* is a list of zero or more additional arguments, which will be formatted according to the format specifiers listed in the format string. The syntax for writing the format specifiers is the same as with the `System.out.printf` method. The method creates a string in memory that is formatted as specified, and returns a reference to that string. For example, look at the program in [Code Listing 2-38](#). The program's output is shown in Figure 3-30.

Code Listing 2-38 (CurrencyFormat2.java)

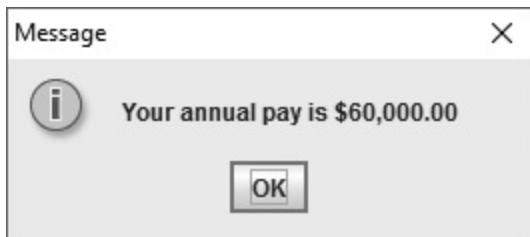
```
1 import javax.swing.JOptionPane;
2
3 /**
4  * This program demonstrates how to use the String.format
5  * method to format a number as currency.
6 */
7
8 public class CurrencyFormat2
9 {
10    public static void main(String[] args)
11    {
12        double monthlyPay = 5000.0;
13        double annualPay = monthlyPay * 12;
14        String output = String.format("Your annual pay is $%, .2f",
15
16        JOptionPane.showMessageDialog(null, output);
17    }
18 }
```

Let's take a closer look at the program. Line 12 declares a double variable named `monthlyPay`, initialized with the value 5000.0, and line 13 declares a double variable named `annualPay`, initialized with the result of the

`calculation monthlyPay * 12.` Line 14 declares a `String` variable named `output`, and initializes it with the string that is returned from the `String.format` method. In line 16, the `output` variable is passed as an argument to the `JOptionPane.showMessageDialog` method.

The program in [Code Listing 2-38](#) can be simplified. We can combine the steps of calling the `String.format` method and passing the value that it returns to the `JOptionPane.showMessageDialog` method. This allows us to eliminate the declaration of the `output` variable. [Code Listing 2-39](#) shows how this is done. The program's output is the same as shown in [Figure 2-25](#).

Figure 2-25 Output of [Code Listing 2-38](#)



Code Listing 2-39 (CurrencyFormat3.java)

```
1 import javax.swing.JOptionPane;
2
3 /**
4  * This program demonstrates how to use the String.format
5  * method to format a number as currency.
6 */
7
8 public class CurrencyFormat3
9 {
10    public static void main(String[] args)
11    {
12        double monthlyPay = 5000.0;
```

```
13     double annualPay = monthlyPay * 12;  
14  
15     JOptionPane.showMessageDialog(null,  
16         String.format("Your annual pay is $%, .2f", annualPay));  
17     }  
18 }
```



Checkpoint

1. 2.38 Assume the following variable declaration exists in a program:

```
double number = 1234567.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable, formatted as:

1,234,567.46

2. 2.39 Assume the following variable declaration exists in a program:

```
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable rounded to one decimal place, in a field that is 10 spaces wide. (Do not use comma separators.)

3. 2.40 Assume the following variable declaration exists in a program:

```
double number = 123.456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable padded with leading zeros, in a field that is eight spaces wide, rounded to one decimal place. (Do not use comma separators.)

4. 2.41 Assume the following variable declaration exists in a program:

```
int number = 123456;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable in a field that is 10 spaces wide, with comma separators.

5. 2.42 Assume the following variable declaration exists in a program:

```
double number = 123456.789;
```

Write a statement that uses `System.out.printf` to display the value of the `number` variable left-justified, with comma separators, in a field that is 20 spaces wide, rounded to two decimal places.

6. 2.43 Assume the following declaration exists in a program:

```
String name = "James";
```

Write a statement that uses `System.out.printf` to display the value of `name` in a field that is 20 spaces wide.

2.16 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- Mismatched braces, quotation marks, or parentheses. In this chapter, you saw that the statements making up a class definition are enclosed in a set of braces. Also, you saw that the statements in a method are enclosed in a set of braces. For every opening brace, there must be a closing brace in the proper location. The same is true of double-quotation marks that enclose string literals, and single quotation marks that enclose character literals. Also, in a statement that uses parentheses (such as a mathematical expression) you must have a closing parenthesis for every opening parenthesis.
- Misspelling key words. Java will not recognize a key word that has been misspelled.
- Using capital letters in key words. Remember that Java is a case-sensitive language, and all key words are written in lowercase. Using an uppercase letter in a key word is the same as misspelling the key word.
- Using a key word as a variable name. The key words are reserved for special uses; they cannot be used for any other purpose.
- Using inconsistent spelling of variable names. Each time you use a variable name, it must be spelled exactly as it appears in its declaration statement.
- Using inconsistent case of letters in variable names. Because Java is a case-sensitive language, it distinguishes between uppercase and lowercase letters. Java will not recognize a variable name that is not written exactly as it appears in its declaration statement.
- Inserting a space in a variable name. Spaces are not allowed in variable names. Instead of using a two-word name such as `gross pay`, use one

word, such as `grossPay`.

- Forgetting the semicolon at the end of a statement. A semicolon appears at the end of each complete statement in Java.
- Assigning a double literal to a `float` variable. Java is a strongly typed language, which means that it only allows you to store values of compatible data types in variables. All floating-point literals are treated as doubles, and a double value is not compatible with a `float` variable. A floating-point literal must end with the letter `f` or `F` in order to be stored in a `float` variable.
- Using commas or other currency symbols in numeric literals. Numeric literals cannot contain commas or currency symbols, such as the dollar sign.
- Unintentionally performing integer division. When both operands of a division statement are integers, the statement will result in an integer. If there is a remainder, it will be discarded.
- Forgetting to group parts of a mathematical expression. If you use more than one operator in a mathematical expression, the expression will be evaluated according to the order of operations. If you wish to change the order in which the operators are used, you must use parentheses to group part of the expression.
- Inserting a space in a combined assignment operator. A space cannot appear between the two operators that make a combined assignment operator.
- Using a variable to receive the result of a calculation when the variable's data type is incompatible with the data type of the result. A variable that receives the result of a calculation must be of a data type that is compatible with the data type of the result.
- Incorrectly terminating a multiline comment or a documentation comment. Multi-line comments and documentation comments are terminated by the `*/` characters. Forgetting to place these characters at a

comment's desired ending point, or accidentally switching the * and the /, will cause the comment not to have an ending point.

- Forgetting to use the correct `import` statement in a program that uses the `Scanner` class or the `JOptionPane` class. In order for the `Scanner` class to be available to your program, you must have the `import java.util.Scanner;` statement near the top of your program file. In order for the `JOptionPane` class to be available to your program, you must have the `import javax.swing.JOptionPane;` statement near the top of the program file.
- When using an input dialog to read numeric input, not converting the `showInputDialog` method's return value to a number. The `showInputDialog` method always returns the user's input as a string. If the user enters a numeric value, it must be converted to a number before it can be used in a math statement.

Review Questions and Exercises

Multiple Choice and True/False

1. Every complete statement ends with a .

1. period
2. parenthesis
3. semicolon
4. ending brace

2. The following data:

```
72  
'A'  
"Hello World"  
2.8712
```

are all examples of .

1. variables
 2. literals
 3. strings
 4. none of these
3. A group of statements, such as the contents of a class or a method, are enclosed in .
1. braces {}

2. parentheses ()
 3. brackets []
 4. any of these will do
4. Which of the following are *not* valid assignment statements? (Indicate all that apply.)
 1. total = 9;
 2. 72 = amount;
 3. profit = 129
 4. letter = 'w';
5. Which of the following are not valid `println` statements? (Indicate all that apply.)
 1. `System.out.println + "Hello World";`
 2. `System.out.println("Have a nice day");`
 3. `out.System.println(value);`
 4. `println.out(Programming is great fun);`
6. The negation operator is .
 1. unary
 2. binary
 3. ternary
 4. none of these
7. Which key word is used to declare a named constant?

1. constant
 2. namedConstant
 3. final
 4. concrete
8. Which characters mark the beginning of a multiline comment?
1. //
 2. /*
 3. */
 4. /**
9. Which characters mark the beginning of a single-line comment?
1. //
 2. /*
 3. */
 4. /**
10. Which characters mark the beginning of a documentation comment?
1. //
 2. /*
 3. */
 4. /**
11. Which Scanner class method would you use to read a string as input?

1. nextString

2. nextLine

3. readString

4. getLine

12. Which Scanner class method would you use to read a double as input?

1. nextDouble

2. getDouble

3. readDouble

4. None of these; you cannot read a double with the Scanner class

13. You can use this class to display dialog boxes.

1. JOptionPane

2. BufferedReader

3. InputStreamReader

4. DialogBox

14. When Java converts a lower-ranked value to a higher-ranked type, it is called a .

1. 4-bit conversion

2. escalating conversion

3. widening conversion

4. narrowing conversion

15. Which type of operator lets you manually convert a value, even if it

means that a narrowing conversion will take place?

1. cast
2. binary
3. uploading
4. dot

16. **True or False:** A left brace in a Java program is always followed by a right brace later in the program.
17. **True or False:** A variable must be declared before it can be used.
18. **True or False:** Variable names may begin with a number.
19. **True or False:** You cannot change the value of a variable whose declaration uses the `final` key word.
20. **True or False:** Comments that begin with `//` can be processed by javadoc.
21. **True or False:** If one of an operator's operands is a `double` and the other operand is an `int`, Java will automatically convert the value of the `double` to an `int`.

Predict the Output

What will the following code segments print on the screen?

1.

```
int freeze = 32, boil = 212;
freeze = 0;
boil = 100;
System.out.println(freeze + "\n"+ boil + "\n");
```
2.

```
int x = 0, y = 2;
x = y * 4;
```

```

System.out.println(x + "\n" + y + "\n");
3. System.out.print("I am the incredible");
   System.out.print("computing\nmachine");
   System.out.print("\nand I will\nnameze\n");
   System.out.println("you.");
4. System.out.print("Be careful\n");
   System.out.print("This might/n be a trick ");
   System.out.println("question.");
5. int a, x = 23;
   a = x % 2;
   System.out.println(x + "\n" + a);

```

Find the Error

There are a number of syntax errors in the following program. Locate as many as you can.

```

/* What's wrong with this program? */
public MyProgram
{
    public static void main(String[] args);
}
int a, b, c  \\ Three integers
a = 3
b = 4
c = a + b
System.out.println('The value of c is' + c);
{

```

Algorithm Workbench

1. Show how the double variables temp, weight, and age can be declared in one statement.
2. Show how the int variables months, days, and years may be declared in one statement, with months initialized to 2 and years initialized to 3.

3. Write assignment statements that perform the following operations with the variables a, b, and c.
1. Adds 2 to a and stores the result in b
 2. Multiplies b times 4 and stores the result in a
 3. Divides a by 3.14 and stores the result in b
 4. Subtracts 8 from b and stores the result in a
 5. Stores the character ‘K’ in c
 6. Stores the Unicode code for ‘B’ in c
4. Assume the variables result, w, x, y, and z are all integers, and that w = 5, x = 4, y = 8, and z = 2. What value will be stored in result in each of the following statements?
1. `result = x + y;`
 2. `result = z * 2;`
 3. `result = y / x;`
 4. `result = y - z;`
 5. `result = w % 2;`
5. How would each of the following numbers be represented in E notation?
1. 3.287×10^6
 2. -9.7865×10^{12}
 3. 7.65491×10^{-3}
6. Modify the following program so it prints two blank lines between each line of text.

```
public class
{
    public static void main(String[] args)
    {
        System.out.print("This is line 1.");
        System.out.print("This is line 2.");
        System.out.print("This is line 3.");
        System.out.print("This is line 4.");
        System.out.println("This is line 5.");
    }
}
```

7. What will the following code output?

```
int apples = 0, bananas = 2, pears = 10;
apples += 10;
bananas *= 10;
pears /= 10; System.out.println(apples + " " +
                    bananas + " " +
                    pears);
```

8. What will the following code output?

```
double d = 12.9;
int i = (int)d;
System.out.println(i);
```

9. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.charAt(5));
```

10. What will the following code output?

```
String message = "Have a great day!";
System.out.println(message.toUpperCase());
System.out.println(message);
```

11. Convert the following pseudocode to Java code. Be sure to declare the appropriate variables.

Store 20 in the speed variable.

Store 10 in the time variable.

Multiply speed by time and store the result in the distance variable.

Display the contents of the distance variable.

12. Convert the following pseudocode to Java code. Be sure to define the appropriate variables.

Store 172.5 in the force variable.

Store 27.5 in the area variable.

Divide area by force and store the result in the pressure variable.

Display the contents of the pressure variable.

13. Write the code to set up all the necessary objects for reading keyboard input. Then write code that asks the user to enter his or her desired annual income. Store the input in a double variable.
14. Write the code to display a dialog box that asks the user to enter his or her desired annual income. Store the input in a double variable.
15. A program has a float variable named total, and a double variable named number. Write a statement that assigns number to total without causing an error when compiled.

Short Answer

1. Is the following comment a single-line style comment or a multiline style comment?

```
/* This program was written by M. A. Codewriter */
```

2. Is the following comment a single-line style comment or a multiline style comment?

```
// This program was written by M. A. Codewriter
```

3. Describe what the term *self-documenting program* means.

4. What is meant by “case sensitive”? Why is it important for a programmer to know that Java is a case-sensitive language?

5. Briefly explain how the `print` and `println` methods are related to the `System` class and the `out` object.
6. What does a variable declaration tell the Java compiler about a variable?
7. Why are variable names like `x` not recommended?
8. What things must be considered when deciding on a data type to use for a variable?
9. Briefly describe the difference between variable assignment and variable initialization.
10. What is the difference between comments that start with the `//` characters, and comments that start with the `/*` characters?
11. Briefly describe what programming style means. Why should your programming style be consistent?
12. Assume that a program uses the named constant `pi` to represent the value 3.14. The program uses the named constant in several statements. What is the advantage of using the named constant instead of the actual value 3.14 in each statement?
13. Assume the file `SalesAverage.java` is a Java source file that contains documentation comments. Assuming you are in the same folder or directory as the source code file, what command would you enter at the operating system command prompt to generate the HTML documentation files?
14. An expression adds a `byte` variable and a `short` variable. Of what data type will the result be?

Programming Challenges

1. Name, Age, and Annual Income

Write a program that declares the following:

- a `String` variable named `name`
- an `int` variable named `age`
- a `double` variable named `annualPay`

Store your age, name, and desired annual income as literals in these variables. The program should display these values on the screen in a manner similar to the following:

My name is Joe Mahoney, my age is 26 and I hope to earn \$1000

2. Name and Initials

Write a program that has the following `String` variables: `firstName`, `middleName`, and `lastName`. Initialize these with your first, middle, and last names. The program should also have the following `char` variables: `firstInitial`, `middleInitial`, and `lastInitial`. Store your first, middle, and last initials in these variables. The program should display the contents of these variables on the screen.

3. Personal Information

Write a program that displays the following information, each on a separate line:

- Your name
- Your address, with city, state, and ZIP
- Your telephone number

- Your college major

Although these items should be displayed on separate output lines, use only a single `println` statement in your program.

4. Star Pattern

Write a program that displays the following pattern:

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

5. Cookie Calories

A bag of cookies holds 40 cookies. The calorie information on the bag claims that there are 10 servings in the bag, and that a serving equals 300 calories. Write a program that lets the user enter the number of cookies he or she actually ate, then reports the number of total calories consumed.

6. Sales Prediction

The East Coast sales division of a company generates 65 percent of total sales. Based on that percentage, write a program that will predict how much the East Coast division will generate if the company has \$8.3 million in sales this year.

Hint: Use the value 0.65 to represent 65 percent.

7. Land Calculation

One acre of land is equivalent to 43,560 square feet. Write a program that calculates the number of acres in a tract of land with 389,767 square feet.

Hint: Divide the size of the tract of land by the size of an acre to get the number of acres.

8. Sales Tax

Write a program that will ask the user to enter the amount of a purchase. The program should then compute the state and county sales tax.

Assume the state sales tax is 5.5 percent and the county sales tax is 2 percent. The program should display the amount of the purchase, the state sales tax, the county sales tax, the total sales tax, and the total of the sale (which is the sum of the amount of purchase plus the total sales tax).

Hint: Use the value 0.02 to represent 2 percent, and 0.055 to represent 5.5 percent.



VideoNote The Miles-per-Gallon Problem

9. Miles-per-Gallon

A car's miles-per-gallon (MPG) can be calculated with the following formula:

$$\text{MPG} = \text{Miles driven} / \text{Gallons of gas consumed}$$

Write a program that asks the user for the number of miles driven and the gallons of gas consumed. It should calculate the car's MPG and display the result on the screen.

10. Test Average

Write a program that asks the user to enter three test scores. The program should display each test score, as well as the average of the scores.

11. Male and Female Percentages

Write a program that asks the user for the number of males and the number of females registered in a class. The program should display the percentage of males and females in the class.

Hint: Suppose there are 8 males and 12 females in a class. There are 20 students in the class. The percentage of males can be calculated as $8 \div 20 = 0.4$, or 40%. The percentage of females can be calculated as $12 \div 20 = 0.6$, or 60%.

12. String Manipulator

Write a program that asks the user to enter the name of his or her favorite city. Use a `String` variable to store the input. The program should display the following:

- The number of characters in the city name
- The name of the city in all uppercase letters
- The name of the city in all lowercase letters
- The first character in the name of the city

13. Restaurant Bill

Write a program that computes the tax and tip on a restaurant bill. The program should ask the user to enter the charge for the meal. The tax should be 7.5 percent of the meal charge. The tip should be 18 percent of the total before adding the tax. Display the meal charge, tax amount, tip amount, and total bill on the screen.

14. Stock Commission

Kathryn bought 1,000 shares of stock at a price of \$25.50 per share. She must pay her stock broker a 2 percent commission for the transaction. Write a program that calculates and displays the following:

- The amount paid for the stock alone (without the commission)

- The amount of the commission
- The total amount paid (for the stock plus the commission)

15. Ingredient Adjuster

A cookie recipe calls for the following ingredients:

- 1.5 cups of sugar
- 1 cup of butter
- 2.75 cups of flour

The recipe produces 48 cookies with these amounts of the ingredients. Write a program that asks the user how many cookies he or she wants to make, then displays the number of cups of each ingredient needed for the specified number of cookies.

16. Energy Drink Consumption

A soft drink company recently surveyed 15,000 of its customers and found that approximately 18 percent of those surveyed purchase one or more energy drinks per week. Of those customers who purchase energy drinks, approximately 58 percent of them prefer citrus-flavored energy drinks. Write a program that displays the following:

- The approximate number of customers in the survey who purchase one or more energy drinks per week
- The approximate number of customers in the survey who prefer citrus-flavored energy drinks

17. Word Game

Write a program that plays a word game with the user. The program should ask the user to enter the following:

- His or her name

- His or her age
- The name of a city
- The name of a college
- A profession
- A type of animal
- A pet’s name

After the user has entered these items, the program should display the following story, inserting the user’s input into the appropriate locations:

There once was a person named **NAME** who lived in **CITY**. At the

18. Stock Transaction Program

Last month, Joe purchased some stock in Acme Software, Inc. Here are the details of the purchase:

- The number of shares that Joe purchased was 1,000.
- When Joe purchased the stock, he paid \$32.87 per share.
- Joe paid his stockbroker a commission that amounted to 2 percent of the amount he paid for the stock.

Two weeks later, Joe sold the stock. Here are the details of the sale:

- The number of shares that Joe sold was 1,000.
- He sold the stock for \$33.92 per share.
- He paid his stockbroker another commission that amounted to 2 percent of the amount he received for the stock.

Write a program that displays the following information:

- The amount of money Joe paid for the stock.
- The amount of commission Joe paid his broker when he bought the stock.
- The amount for which Joe sold the stock.
- The amount of commission Joe paid his broker when he sold the stock.
- Display the amount of profit that Joe made after selling his stock and paying the two commissions to his broker. (If the amount of profit that your program displays is a negative number, then Joe lost money on the transaction.)

19. Planting Grapevines

A vineyard owner is planting several new rows of grapevines, and needs to know how many grapevines to plant in each row. She has determined that after measuring the length of a future row, she can use the following formula to calculate the number of vines that will fit in the row, along with the trellis end-post assemblies that will need to be constructed at each end of the row:

$$V=R-2ES$$

The terms in the formula are:

- V is the number of grapevines that will fit in the row.
- R is the length of the row, in feet.
- E is the amount of space used by an end-post assembly, in feet.
- S is the space between vines, in feet.

Write a program that makes the calculation for the vineyard owner. The program should ask the user to input the following:

- The length of the row, in feet
- The amount of space used by an end-post assembly, in feet
- The amount of space between the vines, in feet

Once the input data has been entered, the program should calculate and display the number of grapevines that will fit in the row.

20. Compound Interest

When a bank account pays compound interest, it pays interest not only on the principal amount that was deposited into the account, but also on the interest that has accumulated over time. Suppose you want to deposit some money into a savings account, and let the account earn compound interest for a certain number of years. The formula for calculating the balance of the account after a specified number of years is:

$$A = P(1 + r)^n t$$

The terms in the formula are:

- A is the amount of money in the account after the specified number of years.
- P is the principal amount that was originally deposited into the account.
- r is the annual interest rate.
- n is the number of times per year that the interest is compounded.
- t is the specified number of years.

Write a program that makes the calculation for you. The program should ask the user to input the following:

- The amount of principal originally deposited into the account

- The annual interest rate paid by the account
- The number of times per year that the interest is compounded (for example, if interest is compounded monthly, enter 12; if interest is compounded quarterly, enter 4)
- The number of years the account will be left to earn interest

Once the input data has been entered, the program should calculate and display the amount of money that will be in the account after the specified number of years.

Note: The user should enter the interest rate as a percentage. For example, 2 percent would be entered as 2, not as .02. The program will then have to divide the input by 100 to move the decimal point to the correct position.

Chapter 3 A First Look at Classes and Objects

Topics

1. [3.1 Classes](#)
2. [3.2 More about Passing Arguments](#)
3. [3.3 Instance Fields and Methods](#)
4. [3.4 Constructors](#)
5. [3.5 A BankAccount Class](#)
6. [3.6 Classes, Variables, and Scope](#)
7. [3.7 Packages and import Statements](#)
8. [3.8 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities](#)
9. [3.9 Common Errors to Avoid](#)

3.1 Classes

Concept:

A class is the blueprint for an object. It specifies the attributes and methods that a particular type of object has. From the class, one or more objects may be created.

Have you ever driven a car? If so, you know that a car is made of a lot of components. A car has a steering wheel, an accelerator pedal, a brake pedal, a gear shifter, a speedometer, and numerous other devices with which the driver interacts. There are also a lot of components under the hood, such as the engine, the battery, the radiator, and so forth. A car is not just one single object, but rather a collection of objects that work together.

This same notion also applies to computer programming. Most programming languages that are used today are object-oriented. When you use an object-oriented language, such as Java, you create programs that are made of objects. In programming, an object isn't a physical device like a steering wheel or a brake pedal. Instead, it's a software component that exists in the computer's memory and performs a specific task. In software, an object has two general capabilities:

- An object can store data. The data stored in an object are commonly called *attributes*, or *fields*.
- An object can perform operations. The operations that an object can perform are called *methods*.

Objects are very important in Java; for example, if you need to store a string in memory, you use a `String` object, and if you need to read input from the keyboard, you use a `Scanner` object. When a program needs the services of a

particular type of object, it creates that object in memory, then calls that object's methods as necessary.

Strings as Objects

[Chapter 2](#) introduced you to the Java primitive data types: byte, short, int, long, char, float, double, and boolean. You use these data types to create variables, which are storage locations in the computer's memory. A primitive data type is called "primitive" because a variable created with a primitive data type has no built-in capabilities other than storing a value.

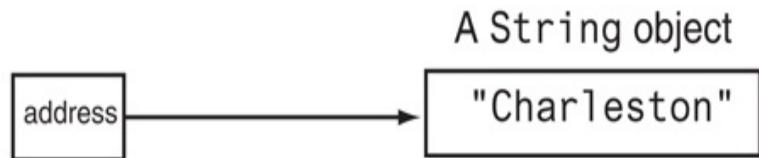
[Chapter 2](#) also introduced you to the `String` class, which allows you to create `String` objects. In addition to storing strings, `String` objects have numerous methods that perform operations on the strings they hold. As a review, let's look at an example. Consider the following statement:

```
String cityName = "Charleston";
```

For each string literal that appears in a Java program, a `String` object is created in memory to hold it. The string literal "Charleston" that appears in this statement causes a `String` object to be created and initialized with the string "Charleston". This statement also declares a variable named `cityName` that references the `String` object. This means that the `cityName` variable holds the `String` object's memory address. This is illustrated in [Figure 3-1](#).

Figure 3-1 The `cityName` variable references a `String` object

The `cityName` variable holds the address of a `String` object.



[Figure 3-1 Full Alternative Text](#)

Assume that the same program has an `int` variable named `stringSize`. Look at the following statement:

```
stringSize = cityName.length();
```

This statement calls the `String` class's `length` method, which returns the length of a string. The expression `cityName.length()` returns the length of the string referenced by `cityName`. After this statement executes, the `stringSize` variable will contain the value 10, which is the length of the string "Charleston".

As you saw in [Chapter 2](#), the `String` class has other methods in addition to `length`. This illustrates one of the differences between an object created from a class, and a variable created from a primitive data type. Class objects normally have methods that perform useful operations on their data. Primitive variables, however, only store data and have no methods. Any operations performed on a primitive variable must be written in code that is external to the variable.

Classes and Instances

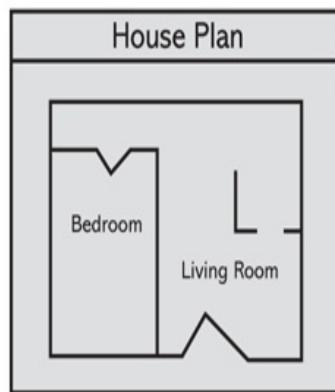
Objects are very useful, but they don't just magically appear in your program. Before a specific type of object can be used by a program, that object has to be created in memory. And before an object can be created in memory, you must have a class for the object.

A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's attributes) and the actions that an object can perform (the object's methods). You can think of a class as a code "blueprint" that can be used to create a particular type of object. It serves a similar purpose as the blueprint for a house. The blueprint itself is not a house, but is a detailed description of a house. When we use the blueprint to build an actual house, we could say we are building an instance of the house described by the blueprint. If we so desire, we can build several identical houses from the same blueprint. Each house is a separate instance of the

house described by the blueprint. This idea is illustrated in [Figure 3-2](#).

Figure 3-2 A blueprint and houses built from the blueprint

Blueprint that describes a house



Instances of the house described by the blueprint



So, a class is not an object, but a description of an object. When a program is running, it can use the class to create as many objects as needed. Each object is considered an *instance* of the class. All of the objects that are created from the same class will have the attributes and methods described by the class.

For example, we can create several objects from the `String` class, as demonstrated with the following code:

```
String person = "Jenny";
String pet = "Fido";
String favoriteColor = "Blue";
```

As illustrated in [Figure 3-3](#), this code creates three `String` objects in memory, which are referenced by the `person`, `pet`, and `favoriteColor` variables.

Figure 3-3 Three variables referencing three String objects

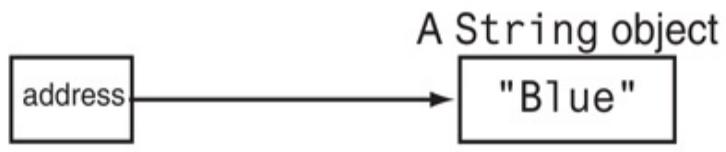
The `person` variable holds the address of a `String` object.



The `pet` variable holds the address of a `String` object.



The `favoriteColor` variable holds the address of a `String` object.



[Figure 3-3 Full Alternative Text](#)

Although each of the three `String` objects holds different data, they are all

identical in design. For example, we can call the `length` method for each of the objects as shown here:

```
stringSize = person.length();
stringSize = pet.length();
stringSize = favoriteColor.length();
```

Because each of the three objects is an instance of the `String` class, each has the attributes and methods specified by the `String` class.

Building a Simple Class Step by Step

In this section, we will write a class named `Rectangle`. Each object that is created from the `Rectangle` class will be able to hold data about a rectangle. Specifically, a `Rectangle` object will have the following attributes:

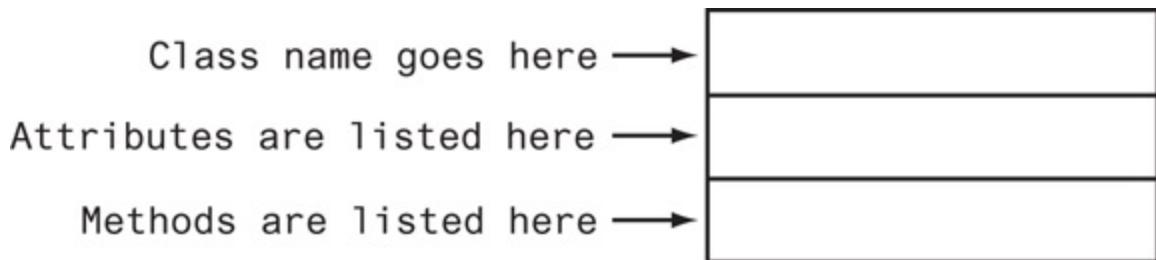
- `length`. The `length` attribute will hold the rectangle's length.
- `width`. The `width` attribute will hold the rectangle's width.

The `Rectangle` class will also have the following methods:

- `setLength`. The `setLength` method will store a value in the `length` attribute.
- `setWidth`. The `setWidth` method will store a value in the `width` attribute.
- `getLength`. The `getLength` method will return the value in the `length` attribute.
- `getWidth`. The `getWidth` method will return the value in the `width` attribute.
- `getArea`. The `getArea` method will return the area of the rectangle, which is the result of its length multiplied by its width.

When designing a class, it is often helpful to draw a UML diagram. *UML* stands for *Unified Modeling Language*. It provides a set of standard diagrams for graphically depicting object-oriented systems. [Figure 3-4](#) shows the general layout of a UML diagram for a class. Notice that the diagram is a box that is divided into three sections. The top section is where you write the name of the class. The middle section holds a list of the class's attributes. The bottom section holds a list of the class's methods.

Figure 3-4 General layout of a UML diagram for a class



[Figure 3-4 Full Alternative Text](#)

Following this layout, [Figure 3-5](#) shows a UML diagram for our Rectangle class. Throughout this book, we will frequently use UML diagrams to illustrate classes.

Figure 3-5 UML diagram for the Rectangle class

Rectangle
length width
setLength() setWidth() getLength() getWidth() getArea()

[Figure 3-5 Full Alternative Text](#)

Writing the Code for a Class

Now that we have identified the attributes and methods that we want the `Rectangle` class to have, let's write the Java code. First, we use an editor to create a new file named `Rectangle.java`. In the `Rectangle.java` file, we will start by writing a general class “skeleton” as follows.



VideoNote Writing Classes and Creating Objects

```
public class Rectangle
{
}
```

The key word `public`, which appears in the first line, is an access specifier. An *access specifier* indicates how the class may be accessed. The `public` access specifier indicates that the class will be publicly available to code that is written outside the `Rectangle.java` file. Almost all of the classes that we will write in this book will be `public`.

Following the access specifier is the key word `class`, followed by `Rectangle`, which is the name of the class. On the next line an opening brace appears, which is followed by a closing brace. The contents of the class,

which are the attributes and methods, will be written inside these braces. The general format of a class definition is:

```
AccessSpecifier class Name
{
    Members
}
```

In general terms, the attributes and methods that belong to a class are referred to as the class's *members*.

Writing the Code for the Class Attributes

Let's continue writing our `Rectangle` class by filling in the code for some of its members. First, we will write the code for the class's two attributes, `length` and `width`. We will use variables of the `double` data type for these attributes. The new lines of code are shown in bold, as follows:

```
public class Rectangle
{
    private double length;
    private double width;
}
```

These two lines of code that we have added declare the variables `length` and `width`. Notice that both declarations begin with the key word `private`, preceding the data type. The key word `private` is an access specifier. It indicates that these variables may not be accessed by statements outside the class.

Recall from our discussion in [Chapter 1](#) on object-oriented programming that an object can perform data hiding, which means that critical data stored inside the object is protected from code outside the object. In Java, a class's private members are hidden, and can be accessed only by methods that are members of the same class. When an object's internal data is hidden from outside code and access to that data is restricted to the object's methods, the

data is protected from accidental corruption. It is a common practice in object-oriented programming to make all of a class's attributes private, and to provide access to those attributes through methods.

When writing classes, you will primarily use the `private` and `public` access specifiers for class members. [Table 3-1](#) summarizes these access specifiers.

Table 3-1 Summary of the private and public Access Specifiers for Class Members

Access Specifier	Description
<code>private</code>	When the <code>private</code> access specifier is applied to a class member, the member cannot be accessed by code outside the class. The member can be accessed only by methods that are members of the same class.
<code>public</code>	When the <code>public</code> access specifier is applied to a class member, the member can be accessed by code inside the class or outside.

You can optionally initialize an attribute with a value. For example, the following statements declare `length` and `width` and initialize them with the values 10 and 12, respectively:

```
private double length = 10;  
private double width = 12;
```

If you do not provide initialization values for numeric attributes, they will be automatically initialized with 0. We will discuss default initialization in greater detail later in this chapter.

Before moving on, we should introduce the term *field*. In Java, a *field* is a class member that holds data. In our `Rectangle` class, the `length` and `width`

variables are both fields.

Tip:

We have referred to `length` and `width` both as attributes and fields. Don't let this confuse you. The term "attribute" is a generic OOP term that refers to an item of data held by an object. The term "field" is a Java-specific term that refers to a member of a class that holds data. In Java, you use fields as attributes.

Writing the `setLength` Method

Now we will begin writing the class methods. We will start with the `setLength` method. This method will allow code outside the class to store a value in the `length` field. [Code Listing 3-1](#) shows the `Rectangle` class at this stage of its development. The `setLength` method is in lines 16 through 19. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find `Rectangle.java` in the folder *Chapter 03\Rectangle Class Phase 1*.)

Code Listing 3-1 (`Rectangle.java`)

```
1 /**
2  * Rectangle class, Phase 1
3  * Under Construction!
4 */
5
6 public class Rectangle
7 {
8     private double length;
9     private double width;
10
11    /**
12     * The setLength method accepts an argument
13     * that is stored in the length field.
14 */
```

```
15  
16  public void setLength(double len)  
17  {  
18      length = len;  
19  }  
20 }
```

In lines 11 through 14, we write a block comment that gives a brief description of the method. It's important to always write comments that describe a class's methods so in the future, anyone reading the code will better understand it. The definition of the method appears after the block comment in lines 16 through 19. The first line of the method definition, which appears in line 16, is known as the *method header*. It appears as:

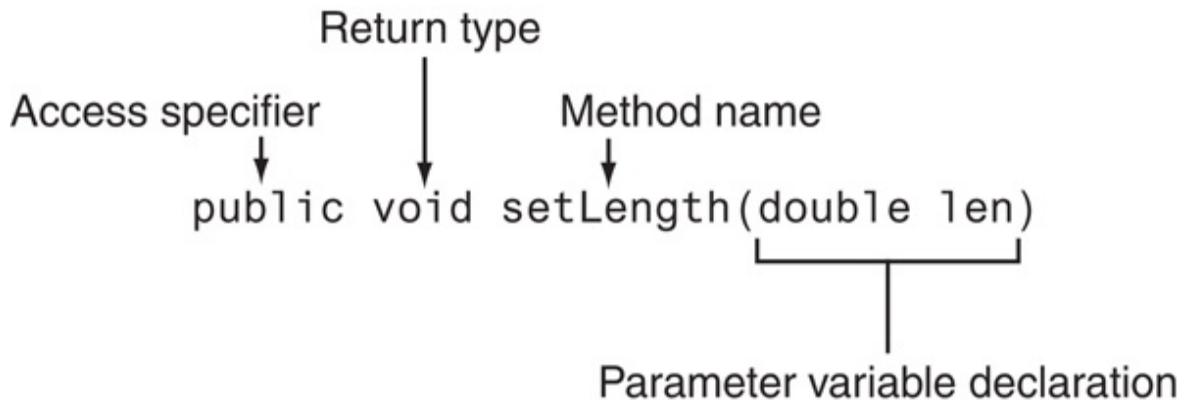
```
public void setLength(double len)
```

The method header has several parts. Let's look at each one.

- `public`. The key word `public` is an access specifier. It indicates that the method may be called by statements outside the class.
- `void`. This is the method's return type. The key word `void` indicates that the method returns no data to the statement that called it.
- `setLength`. This is the name of the method.
- `(double len)`. This is the declaration of a parameter variable. A parameter variable holds the value of an argument that is passed to the method. The parameter variable's name is `len`, and it is of the `double` data type.

[Figure 3-6](#) labels each part of the header for the `setLength` method.

Figure 3-6 Header for the `setLength` method



[Figure 3-6 Full Alternative Text](#)

After the header, the body of the method appears inside a set of braces:

```
{
    length = len;
}
```

This method has only one statement, which assigns the value of `len` to the `length` field. When the method executes, the `len` parameter variable will hold the value of an argument that is passed to the method. That value is assigned to the `length` field.

Before adding the other methods to the class, it might help if we demonstrate how the `setLength` method works. First, notice that the `Rectangle` class does not have a `main` method. This class is not a complete program, but is a blueprint from which `Rectangle` objects may be created. Other programs will use the `Rectangle` class to create objects. The programs that create and use these objects will have their own `main` methods. We can demonstrate the class's `setLength` method by saving the current contents of the `Rectangle.java` file, then creating the program shown in [Code Listing 3-2](#). (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03/Rectangle Class Phase 1*.)

Code Listing 3-2 (LengthDemo.java)

```

1 /**
2  * This program demonstrates the Rectangle class's
3  * setLength method.
4 */
5
6 public class LengthDemo
7 {
8     public static void main(String[] args)
9     {
10     Rectangle box = new Rectangle();
11
12     System.out.println("Sending the value 10.0 to " +
13                     "the setLength method.");
14     box.setLength(10.0);
15     System.out.println("Done.");
16 }
17 }
```

The program in [Code Listing 3-2](#) must be saved as *LengthDemo.java* in the same folder or directory as the file *Rectangle.java*. The following command can then be used to compile the program:

```
javac LengthDemo.java
```

When the compiler reads the source code for *LengthDemo.java* and sees that a class named *Rectangle* is being used, it looks in the current folder or directory for the file *Rectangle.class*. That file does not exist, however, because we have not yet compiled *Rectangle.java*. So, the compiler searches for the file *Rectangle.java* and compiles it. This creates the file *Rectangle.class*, which makes the *Rectangle* class available. The compiler then finishes compiling *LengthDemo.java*. The resulting *LengthDemo.class* file may be executed with the following command:

```
java LengthDemo
```

The output of the program is as follows.

Program Output

```
Sending the value 10.0 to the setLength method.
Done.
```

Let's look at each statement in this program's *main* method. In line 10, the

program uses the following statement to create a `Rectangle` class object and associate it with a variable:

```
Rectangle box = new Rectangle();
```

Let's dissect the statement into two parts. The first part of the statement,

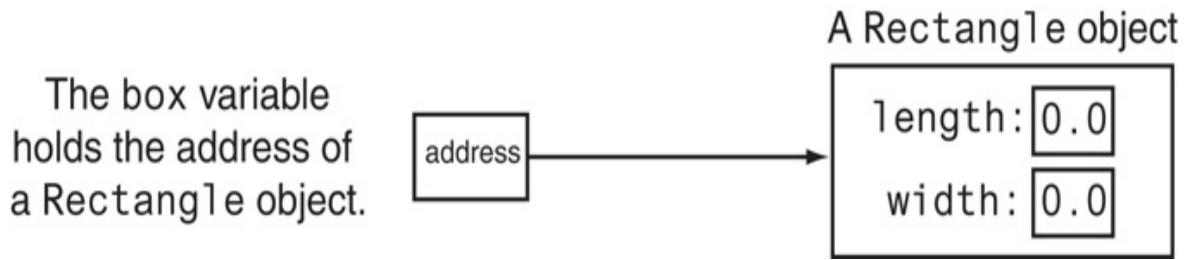
```
Rectangle box
```

declares a variable named `box`. The data type of the variable is `Rectangle`. (Because the word `Rectangle` is not the name of a primitive data type, Java assumes it to be the name of a class.) Recall from [Chapter 2](#) that a variable of a class type is a reference variable, and it holds the memory address of an object. When a reference variable holds an object's memory address, it is said that the variable references the object. So, the variable `box` will be used to reference a `Rectangle` object. The second part of the statement is:

```
= new Rectangle();
```

This part of the statement uses the key word `new`, which creates an object in memory. After the word `new`, the name of a class followed by a set of parentheses appears. This specifies the class that the object should be created from. In this case, an object of the `Rectangle` class is created. The memory address of the object is then assigned (by the `=` operator) to the variable `box`. After the statement executes, the variable `box` will reference the object that was created in memory. This is illustrated in [Figure 3-7](#).

Figure 3-7 The `box` variable references a `Rectangle` class object



[Figure 3-7 Full Alternative Text](#)

Notice that [Figure 3-7](#) shows the `Rectangle` object's `length` and `width` fields set to 0. All of a class's numeric fields are initialized to 0 by default.

Tip:

The parentheses in this statement are required. It would be an error to write the statement as

```
Rectangle box = new Rectangle; // ERROR!!
```

Lines 12 and 13 call the `println` method to display a message on the screen.

```
System.out.println("Sending the value 10.0 to " +
    "the setLength method.");
```

Line 14 calls the `box` object's `setLength` method. As you have already seen from our examples with the `String` class, you use the dot operator (a period) to access the members of a class object. Recall from [Chapter 2](#) that the general form of a method call is

```
refVariable.method(arguments...)
```

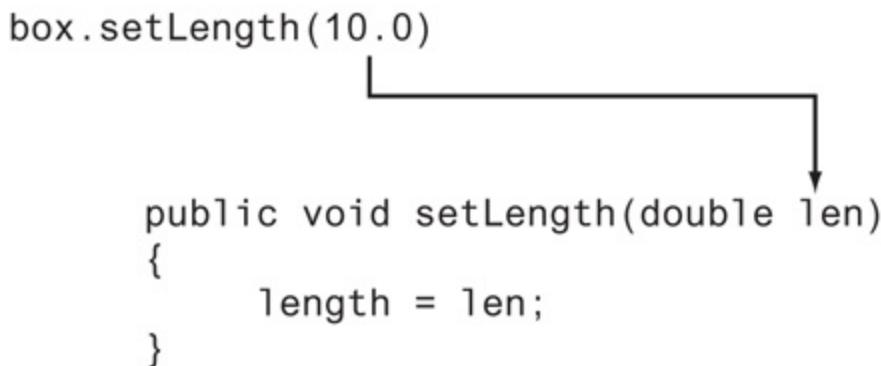
where `refVariable` is the name of a variable that references an object, `method` is the name of a method, and `arguments...` is zero or more arguments that are passed to the method. An argument is a value that is passed into a method. Here is the statement in line 14 that calls the `setLength` method:

```
box.setLength(10.0);
```

This statement passes the argument 10.0 to the `setLength` method. When the

method executes, the value 10.0 is copied into the `len` parameter variable. This is illustrated in [Figure 3-8](#).

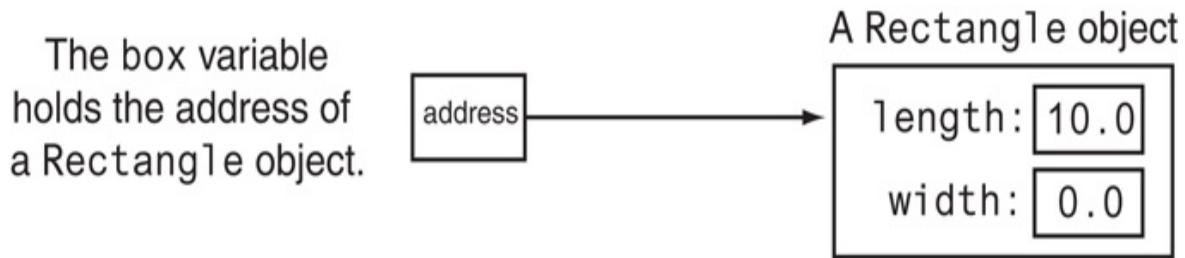
Figure 3-8 The argument 10.0 is copied into the `len` parameter variable



[Figure 3-8 Full Alternative Text](#)

In the `setLength` method, the parameter variable `len` contains the value 10.0. The method assigns the value of `len` to the `length` field and then terminates. [Figure 3-9](#) shows the state of the `box` object after the method executes.

Figure 3-9 The state of the box object after the `setLength` method executes



[Figure 3-9 Full Alternative Text](#)

When passing an argument to a method, the argument's data type must be compatible with the parameter variable's data type. Otherwise, an error will occur. For example, the `len` parameter in the `setLength` method is a double variable. You cannot pass an argument that cannot be automatically converted to the double data type. So, the following statement would cause an error because the argument is a string:

```
box.setLength("10.0"); // ERROR!
```

Writing the `setWidth` Method

Now that we've seen how the `setLength` method works, let's add the `setWidth` method to the `Rectangle` class. The `setWidth` method is similar to `setLength`. It accepts an argument, which is assigned to the `width` field. [Code Listing 3-3](#) shows the updated `Rectangle` class. The `setWidth` method is in lines 26 through 29. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find `Rectangle.java` in the folder *Chapter 03/ Rectangle Class Phase 2*.)

Code Listing 3-3 (`Rectangle.java`)

```

1 /**
2  * Rectangle class, Phase 2
3  * Under Construction!
4 */
5
6 public class Rectangle
7 {

```

```

8  private double length;
9  private double width;
10
11 /**
12 * The setLength method accepts an argument
13 * that is stored in the length field.
14 */
15
16 public void setLength(double len)
17 {
18     length = len;
19 }
20
21 /**
22 * The setWidth method accepts an argument
23 * that is stored in the width field.
24 */
25
26 public void setWidth(double w)
27 {
28     width = w;
29 }
30 }
```

The `setWidth` method has a parameter variable named `w`. When an argument is passed to the method, the argument's value is copied into the `w` variable. The value of the `w` variable is then assigned to the `width` field. For example, assume that `box` references a `Rectangle` object and the following statement is executed:

```
box.setWidth(20.0);
```

After this statement executes, the `box` object's `width` field will be set to 20.0.

Writing the `getLength` and `getWidth` Methods

Because the `length` and `width` fields are private, we wrote the `setLength` and `setWidth` methods to allow code outside the `Rectangle` class to store values in the fields. We must also write methods that allow code outside the class to

get the values that are stored in these fields. That's what the `getLength` and `getWidth` methods will do. The `getLength` method will return the value stored in the `length` field, and the `getWidth` method will return the value stored in the `width` field.

Here is the code for the `getLength` method:

```
public double getLength()
{
    return length;
}
```

Notice that instead of the word `void`, the header uses the word `double` for the method's return type. This means that the method returns a value of the `double` data type. Also, notice that no parameter variables are declared inside the parentheses. This means that the method does not accept arguments. The parentheses are still required, however.

Inside the method, the following statement appears:

```
return length;
```

This is called a *return statement*. The value that appears after the key word `return` is sent back to the statement that called the method. This statement sends the value that is stored in the `length` field. For example, assume that `size` is a `double` variable, and `box` references a `Rectangle` object, and the following statement is executed:

```
size = box.getLength();
```

This statement assigns the value that is returned from the `getLength` method to the `size` variable. After this statement executes, the `size` variable will contain the same value as the `box` object's `length` field. This is illustrated in [Figure 3-10](#).

Figure 3-10 The value returned from `getLength` is assigned to

size

```
size = box.getLength();  
↑  
public double getLength()  
{  
    return length;  
}
```

[Figure 3-10 Full Alternative Text](#)



Note:

No arguments are passed to the `getLength` method. You must still write the parentheses, however, even when no arguments are passed.

The `getWidth` method is similar to `getLength`. The code for the method follows.

```
public double getWidth()  
{  
    return width;  
}
```

This method returns the value that is stored in the `width` field. For example, assume that `size` is a double variable and `box` references a `Rectangle` object, and the following statement is executed:

```
size = box.getWidth();
```

This statement assigns the value that is returned from the `getWidth` method to the `size` variable. After this statement executes, the `size` variable will contain the same value as the `box` object's `width` field.

[Code Listing 3-4](#) shows the Rectangle class with all of the members we have discussed so far. The code for the getLength and getWidth methods is shown in lines 31 through 49. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find *Rectangle.java* in the folder *Chapter 03\Rectangle Class Phase 3*.)

Code Listing 3-4 (Rectangle.java)

```
1 /**
2  * Rectangle class, Phase 3
3  * Under Construction!
4 */
5
6 public class Rectangle
7 {
8     private double length;
9     private double width;
10
11    /**
12     * The setLength method accepts an argument
13     * that is stored in the length field.
14    */
15
16    public void setLength(double len)
17    {
18        length = len;
19    }
20
21    /**
22     * The setWidth method accepts an argument
23     * that is stored in the width field.
24    */
25
26    public void setWidth(double w)
27    {
28        width = w;
29    }
30
31    /**
32     * The getLength method returns the value
33     * stored in the length field.
34    */
35
```

```

36     public double getLength()
37     {
38         return length;
39     }
40
41     /**
42      * The getWidth method returns the value
43      * stored in the width field.
44     */
45
46     public double getWidth()
47     {
48         return width;
49     }
50 }
```

Before continuing, we should demonstrate how these methods work. Look at the program in [Code Listing 3-5](#). (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find the file in the folder *Chapter 03\Rectangle Class Phase 3.*)

Code Listing 3-5 (LengthWidthDemo.java)

```

1  /**
2   * This program demonstrates the Rectangle class's
3   * setLength, setWidth, getLength, and getWidth methods.
4   */
5
6 public class LengthWidthDemo
7 {
8     public static void main(String[] args)
9     {
10     Rectangle box = new Rectangle();
11
12     box.setLength(10.0);
13     box.setWidth(20.0);
14     System.out.println("The box's length is " +
15                         box.getLength());
16     System.out.println("The box's width is " +
17                         box.getWidth());
18 }
19 }
```

Program Output

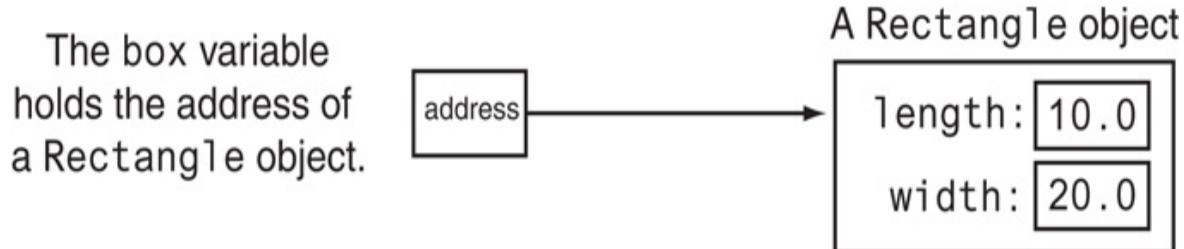
```
The box's length is 10.0  
The box's width is 20.0
```

Let's take a closer look at the program. First, this program creates a `Rectangle` object, which is referenced by the `box` variable. Then the following statements, in lines 12 and 13, execute:

```
box.setLength(10.0);  
box.setWidth(20.0);
```

After these statements execute, the `box` object's `length` field is set to 10.0, and its `width` field is set to 20.0. The state of the object is shown in [Figure 3-11](#).

Figure 3-11 State of the box object



[Figure 3-11 Full Alternative Text](#)

Next, the following statement, in lines 14 and 15, executes:

```
System.out.println("The box's length is " +  
    box.getLength());
```

This statement calls the `box.getLength()` method, which returns the value 10.0. The following message is displayed on the screen:

```
The box's length is 10.0
```

Then, the statement in lines 16 and 17 executes.

```
System.out.println("The box's width is " +  
    box.getWidth());
```

This statement calls the `box.getWidth()` method, which returns the value 20.0. The following message is displayed on the screen:

```
The box's width is 20.0
```

Writing the `getArea` Method

The last method we will write for the `Rectangle` class is `getArea`. This method returns the area of a rectangle, which is its length multiplied by its width. Here is the code for the `getArea` method:

```
public double getArea()  
{  
    return length * width;  
}
```

This method returns the result of the mathematical expression `length * width`. For example, assume that `area` is a `double` variable, and `box` references a `Rectangle` object, and the following code is executed:

```
box.setLength(10.0);  
box.setWidth(20.0);  
area = box.getArea();
```

The last statement assigns the value that is returned from the `getArea` method to the `area` variable. After this statement executes, the `area` variable will contain the value 200.0.

[Code Listing 3-6](#) shows the `Rectangle` class with all of the members we have discussed so far. The `getArea` method appears in lines 56 through 59. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03\Rectangle Class Phase 4.*)

Code Listing 3-6 (Rectangle.java)

```
1 /**
2  * Rectangle class, Phase 4
3  * Under Construction!
4 */
5
6 public class Rectangle
7 {
8     private double length;
9     private double width;
10
11    /**
12     * The setLength method accepts an argument
13     * that is stored in the length field.
14    */
15
16    public void setLength(double len)
17    {
18        length = len;
19    }
20
21    /**
22     * The setWidth method accepts an argument
23     * that is stored in the width field.
24    */
25
26    public void setWidth(double w)
27    {
28        width = w;
29    }
30
31    /**
32     * The getLength method returns the value
33     * stored in the length field.
34    */
35
36    public double getLength()
37    {
38        return length;
39    }
40
41    /**
42     * The getWidth method returns the value
43     * stored in the width field.

```

```

44     */
45
46     public double getWidth()
47     {
48         return width;
49     }
50
51     /**
52      * The getArea method returns the value of the
53      * length field times the width field.
54     */
55
56     public double getArea()
57     {
58         return length * width;
59     }
60 }
```

The program in [Code Listing 3-7](#) demonstrates all the methods of the Rectangle class, including getArea. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03\Rectangle Class Phase 4*.)

Code Listing 3-7 (**RectangleDemo.java**)

```

1 /**
2  * This program demonstrates the Rectangle class's
3  * setLength, setWidth, getLength, getWidth, and
4  * getArea methods.
5 */
6
7 public class RectangleDemo
8 {
9     public static void main(String[] args)
10    {
11        // Create a Rectangle object.
12        Rectangle box = new Rectangle();
13
14        // Set the length to 10 and width to 20.
15        box.setLength(10.0);
16        box.setWidth(20.0);
17    }
}
```

```
18 // Display the length, width, and area.  
19 System.out.println("The box's length is " +  
20     box.getLength());  
21 System.out.println("The box's width is " +  
22     box.getWidth());  
23 System.out.println("The box's area is " +  
24     box.getArea());  
25 }  
26 }
```

Program Output

```
The box's length is 10.0  
The box's width is 20.0  
The box's area is 200.0
```

Accessor and Mutator Methods

As mentioned earlier, it is a common practice to make all of a class's fields private, and to provide public methods for accessing and changing those fields. This ensures that the object owning those fields is in control of all changes being made to them. A method that gets a value from a class's field, but does not change it, is known as an *accessor method*. A method that stores a value in a field, or in some other way changes the value of a field, is known as a *mutator method*. In the Rectangle class, the methods `getLength` and `getWidth` are accessors, and the methods `setLength` and `setWidth` are mutators.



Note:

Mutator methods are sometimes called “setters”, and accessor methods are sometimes called “getters.”

The Importance of Data Hiding

Data hiding is an important concept in object-oriented programming. An object hides its internal data from code that is outside the class of which the object is an instance. Only the class's methods may directly access and make changes to the object's internal data. You hide an object's internal data by making the class's fields `private` and making the methods that access those fields `public`.

As a beginning student, you might be wondering why you would want to hide the data that is inside the classes you create. As you learn to program, you will be the user of your own classes, so it might seem that you are putting forth a great effort to hide data from yourself. If you write software in industry, however, the classes that you create will be used as components in large software systems, and programmers other than yourself will be using your classes. By hiding a class's data, and allowing it to be accessed only through the class's methods, you can better ensure that the class will operate as you intended it to.

Avoiding Stale Data

Recall that the `Rectangle` class has the methods `getLength`, `getWidth`, and `getArea`. The `getLength` and `getWidth` methods return the values stored in fields, but the `getArea` method returns the result of a calculation. You might be wondering why the area of the rectangle is not stored in a field, like the length and the width. The area is not stored in a field because it could potentially become stale. When the value of an item is dependent on other data, and that item is not updated when the other data is changed, it is said that the item has become *stale*. If the area of the rectangle were stored in a field, the value of the field would become incorrect as soon as either the `length` or `width` field changed.

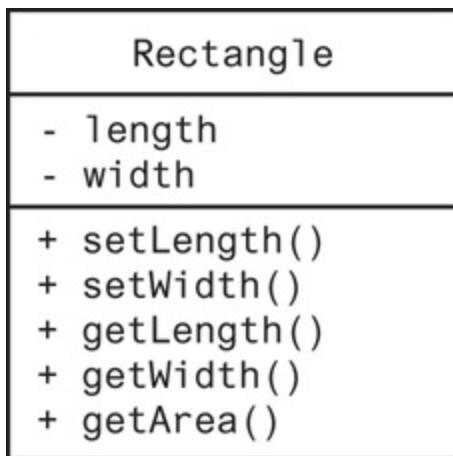
When designing a class, you should take care not to store in a field calculated data that could potentially become stale. Instead, provide a method that returns the result of the calculation.

Showing Access Specification in

UML Diagrams

In [Figure 3-5](#), we presented a UML diagram for the `Rectangle` class. The diagram listed all of the members of the class, but did not indicate which members were private and which were public. In a UML diagram, you have the option to place a `-` character before a member name to indicate that it is private, or a `+` character to indicate that it is public. [Figure 3-12](#) shows the UML diagram modified to include this notation.

Figure 3-12 UML diagram for the `Rectangle` class



[Figure 3-12 Full Alternative Text](#)

Data Type and Parameter Notation in UML Diagrams

The Unified Modeling Language also provides notation that you can use to indicate the data types of fields, methods, and parameters. To indicate the data type of a field, place a colon followed by the name of the data type after

the name of the field. For example, the `length` field in the `Rectangle` class is a double. It could be listed as follows in the UML diagram:

```
- length : double
```

The return type of a method can be listed in the same manner: After the method's name, place a colon followed by the return type. The `Rectangle` class's `getLength` method returns a double, so it could be listed as follows in the UML diagram:

```
+ getLength() : double
```

Parameter variables and their data types may be listed inside a method's parentheses. For example, the `Rectangle` class's `setLength` method has a double parameter named `len`, so it could be listed as follows in the UML diagram:

```
+ setLength(len : double) : void
```

[Figure 3-13](#) shows a UML diagram for the `Rectangle` class with parameter and data type notation.

Figure 3-13 UML diagram for the `Rectangle` class with parameter and data type notation

Rectangle
- length : double - width : double
+ setLength(len : double) : void + setWidth(w : double) : void + getLength() : double + getWidth() : double + getArea() : double

[Figure 3-13 Full Alternative Text](#)

Layout of Class Members

Notice that in the Rectangle class, the field variables are declared first then the methods are defined. You are not required to write field declarations before the method definitions. In fact, some programmers prefer to write the definitions for the public methods first, then write the declarations for the private fields last. Regardless of which style you use, you should be consistent. In this book, we always write the field declarations first, followed by the method definitions. [Figure 3-14](#) shows this layout.

Figure 3-14 Typical layout of class members

```
public class ClassName
{
    
}
```

[Figure 3-14 Full Alternative Text](#)

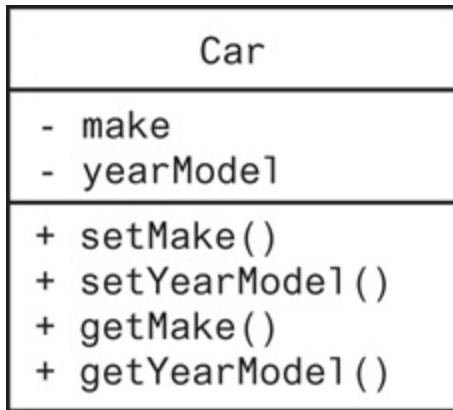


Checkpoint

1. 3.1 In this chapter, we use the metaphor of a blueprint and houses that are created from the blueprint to describe classes and objects. In this metaphor, are classes the blueprint or the houses?
2. 3.2 Describe a way that variables of the primitive data types are different from objects.
3. 3.3 When a variable is said to reference an object, what is actually stored in the variable?
4. 3.4 A string literal, such as “Joe”, causes what type of object to be created?
5. 3.5 Look at the UML diagram in [Figure 3-15](#) and answer the following questions:
 1. What is the name of the class?
 2. What are the attributes?
 3. What are the methods?

4. What are the private members?
5. What are the public members?
6. 3.6 Assume that `limo` is a variable that references an instance of the class depicted in [Figure 3-15](#). Write a statement that calls `setMake` and passes the argument “Cadillac”.
7. 3.7 What does the key word `new` do?
8. 3.8 What is a parameter variable?
9. 3.9 What is an accessor? What is a mutator?
10. 3.10 What is a stale data item?

Figure 3-15 UML diagram



[Figure 3-15 Full Alternative Text](#)

3.2 More about Passing Arguments

Concept:

A method can have multiple parameter variables, allowing you to pass multiple arguments to the method. When an argument is passed to a method, it is passed by value. This means that the parameter variable holds a copy of the value passed to it. Changes made to the parameter variable do not affect the argument.

Passing Multiple Arguments

Often it is useful to pass more than one argument into a method. For example, the `Rectangle` class has two separate methods for setting the `length` and `width` fields: `setLength` and `setWidth`. Setting the `length` and `width` fields by using these methods requires two method calls. We could add another method that accepts two arguments, one for the length and one for the width, making it possible to set both the `length` and the `width` fields with one method call. Here is such a method:

```
public void set(double len, double w)
{
    length = len;
    width = w;
}
```

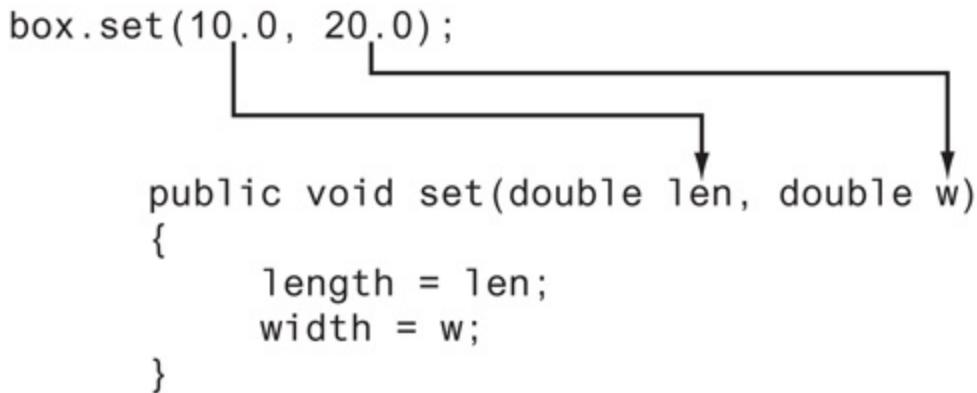
Two parameter variables, `len` and `w`, are declared inside the parentheses in the method header. This requires us to pass two arguments to the method when we call it. For example, assume that `box` references a `Rectangle` object, and the following statement is executed.

```
box.set(10.0, 20.0);
```

This statement passes the value 10.0 into the `len` parameter and the value

20.0 into the w parameter, as illustrated in [Figure 3-16](#).

Figure 3-16 Multiple arguments passed to the set method



[Figure 3-16 Full Alternative Text](#)

Notice that the arguments are passed to the parameter variables in the order in which they appear the method call. In other words, the first argument is passed into the first parameter, and the second argument is passed into the second parameter. For example, the following method call would pass 15.0 into the `len` parameter and 30.0 into the `w` parameter:

```
box.set(15.0, 30.0);
```

The program in [Code Listing 3-8](#) demonstrates passing two arguments to the `set` method. In this program, variables that are declared in the `main` method are passed as the arguments. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file, along with a modified version of the `Rectangle` class, in the folder *Chapter 03\Rectangle Class Phase 5.*)

Code Listing 3-8

(MultipleArgs.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates how to pass
5  * multiple arguments to a method.
6  */
7
8 public class MultipleArgs
9 {
10    public static void main(String[] args)
11    {
12        double boxLength, // To hold the box's length
13            boxWidth; // To hold the box's width
14
15        // Create a Scanner object for keyboard input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Create a Rectangle object.
19        Rectangle box = new Rectangle();
20
21        // Get the box's length.
22        System.out.print("What is the box's length? ");
23        boxLength = keyboard.nextDouble();
24
25        // Get the box's width.
26        System.out.print("What is the box's width? ");
27        boxWidth = keyboard.nextDouble();
28
29        // Pass boxLength and boxWidth to the set method.
30        box.set(boxLength, boxWidth);
31
32        // Display the box's length, width, and area.
33        System.out.println("The box's length is " +
34                    box.getLength());
35        System.out.println("The box's width is " +
36                    box.getWidth());
37        System.out.println("The box's area is " +
38                    box.getArea());
39    }
40 }
```

Program Output with Example Input Shown in Bold

```
What is the box's length? 10.0 Enter
What is the box's width? 20.0 Enter
The box's length is 10.0
The box's width is 20.0
The box's area is 200.0
```

In the program, the user enters values that are stored in the `boxLength` and `boxWidth` variables. The following statement, in line 30, calls the `set` method, passing the `boxLength` and `boxWidth` variables as arguments.

```
box.set(boxLength, boxWidth);
```

When this statement executes, the value stored in the `boxLength` variable is passed into the `len` parameter, and the value stored in the `boxWidth` variable is passed into the `w` parameter.

Arguments Are Passed by Value

In Java, all arguments of the primitive data types are *passed by value*, which means that a copy of an argument's value is passed into a parameter variable. A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call. If a parameter variable is changed inside a method, it has no effect on the original argument.

3.3 Instance Fields and Methods

Concept:

Each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class, and store different values in each instance's fields. The methods that operate on an instance of a class are known as instance methods.

The program in [Code Listing 3-7](#) creates one instance of the `Rectangle` class. It is possible to create many instances of the same class, each with its own data. For example, the `RoomAreas.java` program in [Code Listing 3-9](#) creates three instances of the `Rectangle` class, referenced by the variables `kitchen`, `bedroom`, and `den`.

Code Listing 3-9 (RoomAreas.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program creates three instances of the
5  * Rectangle class.
6 */
7
8 public class RoomAreas
9 {
10    public static void main(String[] args)
11    {
12        double number,      // To hold numeric input
13        totalArea; // The total area of all rooms
14
15        // Create a Scanner object for keyboard input.
16        Scanner keyboard = new Scanner(System.in);
17    }
}
```

```

18 // Create three Rectangle objects.
19 Rectangle kitchen = new Rectangle();
20 Rectangle bedroom = new Rectangle();
21 Rectangle den = new Rectangle();
22
23 // Get and store the dimensions of the kitchen.
24 System.out.print("What is the kitchen's length? ");
25 number = keyboard.nextDouble();
26 kitchen.setLength(number);
27 System.out.print("What is the kitchen's width? ");
28 number = keyboard.nextDouble();
29 kitchen.setWidth(number);
30
31 // Get and store the dimensions of the bedroom.
32 System.out.print("What is the bedroom's length? ");
33 number = keyboard.nextDouble();
34 bedroom.setLength(number);
35 System.out.print("What is the bedroom's width? ");
36 number = keyboard.nextDouble();
37 bedroom.setWidth(number);
38
39 // Get and store the dimensions of the den.
40 System.out.print("What is the den's length? ");
41 number = keyboard.nextDouble();
42 den.setLength(number);
43 System.out.print("What is the den's width? ");
44 number = keyboard.nextDouble();
45 den.setWidth(number);
46
47 // Calculate the total area of the rooms.
48 totalArea = kitchen.getArea() + bedroom.getArea() +
49     den.getArea();
50
51 // Display the total area of the rooms.
52 System.out.println("The total area of the rooms is " +
53     totalArea);
54 }
55 }

```

Program Output with Example Input Shown in Bold

What is the kitchen's length? **10** 

What is the kitchen's width? **14** 

What is the bedroom's length? **15** 

What is the bedroom's width? **12** 

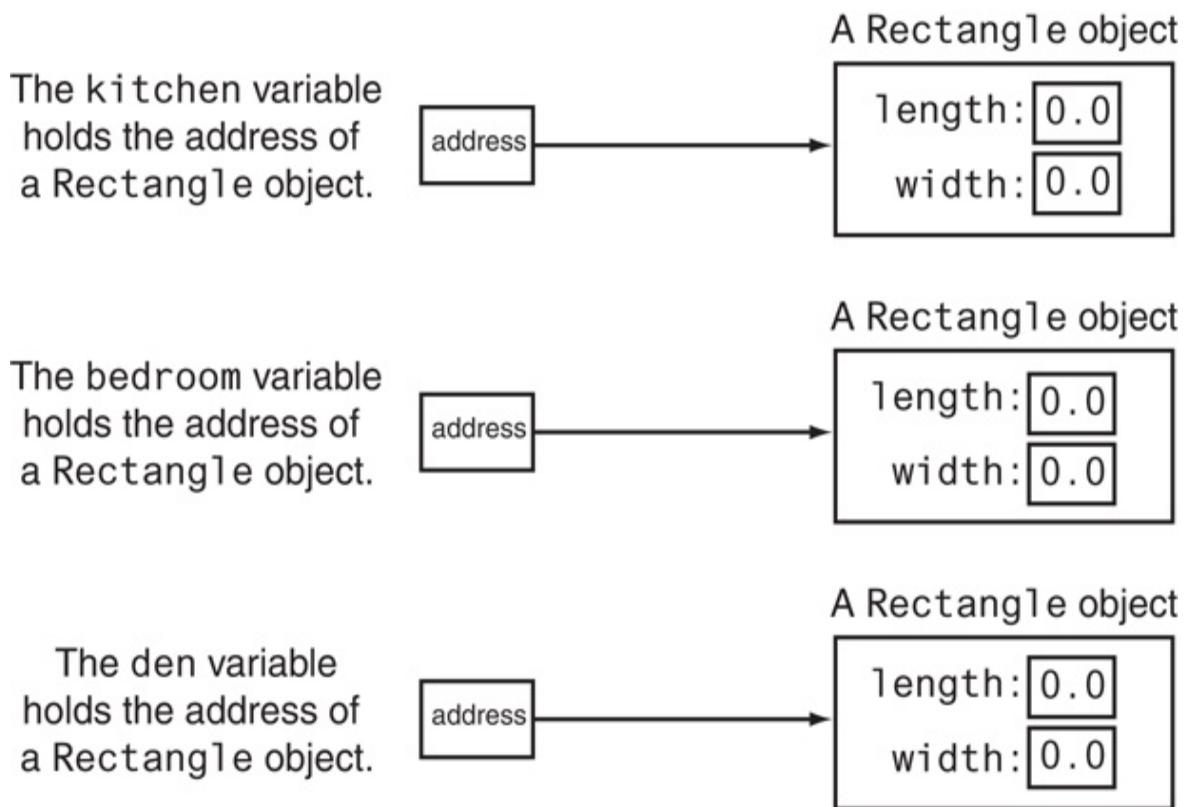
```
What is the den's length? 20 
What is the den's width? 30 
The total area of the rooms is 920.0
```

In the program, the code in lines 19 through 21 creates three objects, each an instance of the Rectangle class:

```
Rectangle kitchen = new Rectangle();
Rectangle bedroom = new Rectangle();
Rectangle den = new Rectangle();
```

[Figure 3-17](#) illustrates how the kitchen, bedroom, and den variables reference the objects.

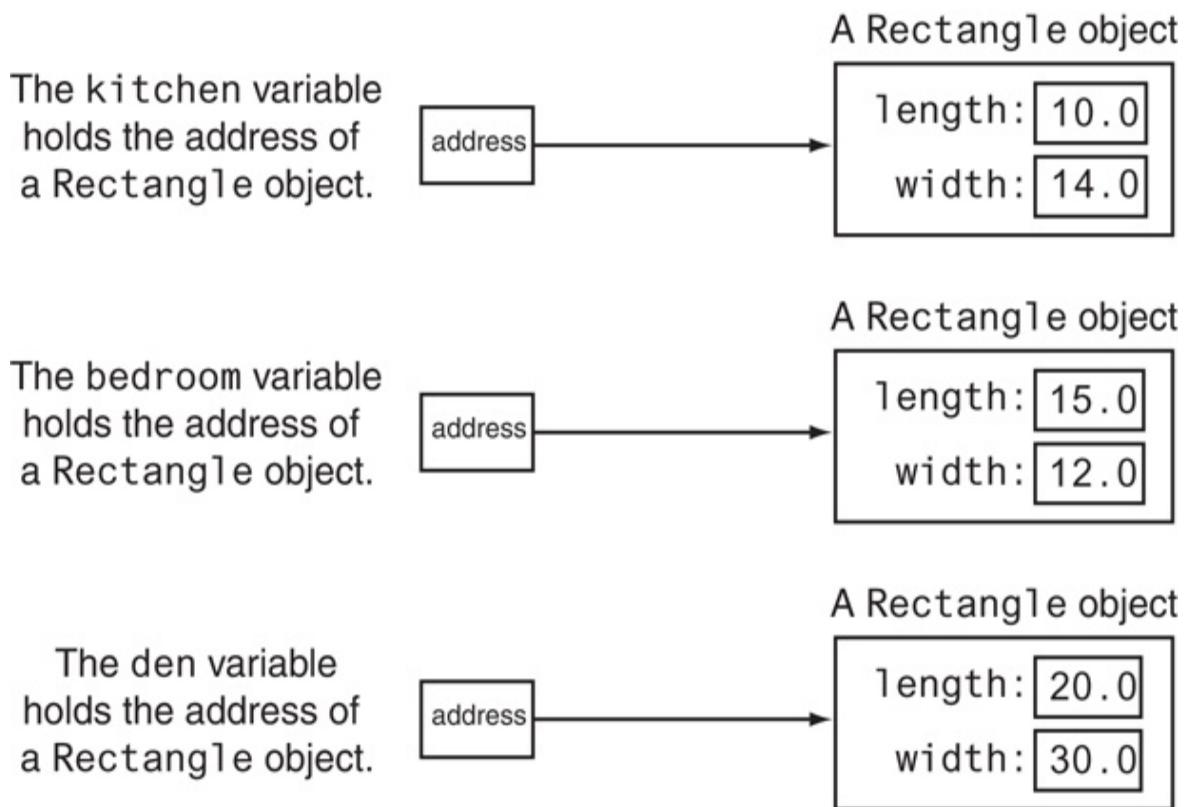
Figure 3-17 The kitchen, bedroom, and den variables reference Rectangle objects



[Figure 3-17 Full Alternative Text](#)

In the example session with the program, the user enters 10 and 14 as the length and width of the kitchen, 15 and 12 as the length and width of the bedroom, and 20 and 30 as the length and width of the den. [Figure 3-18](#) shows the states of the objects after these values are stored in them.

Figure 3-18 States of the objects after data has been stored in them



[Figure 3-18 Full Alternative Text](#)

Notice from [Figure 3-18](#) that each instance of the `Rectangle` class has its own `length` and `width` variables. For this reason, the variables are known as *instance variables*, or *instance fields*. Every instance of a class has its own set of instance fields and can store its own values in those fields.

The methods that operate on an instance of a class are known as *instance methods*. All of the methods in the `Rectangle` class are instance methods because they perform operations on specific instances of the class. For example, look at line 26 in the `RoomsAreas.java` program ([Code Listing 3-9](#)):

```
kitchen.setLength(number);
```

This statement calls the `setLength` method that stores a value in the `kitchen` object's `length` field. Now, look at line 34 in the same program:

```
bedroom.setLength(number);
```

This statement also calls the `setLength` method, but this time it stores a value

in the `bedroom` object's `length` field. Likewise, line 42 calls the `setLength` method to store a value in the `den` object's `length` field:

```
den.setLength(number);
```

The `setLength` method stores a value in a specific instance of the `Rectangle` class. This is true of all of the methods that are members of the `Rectangle` class.



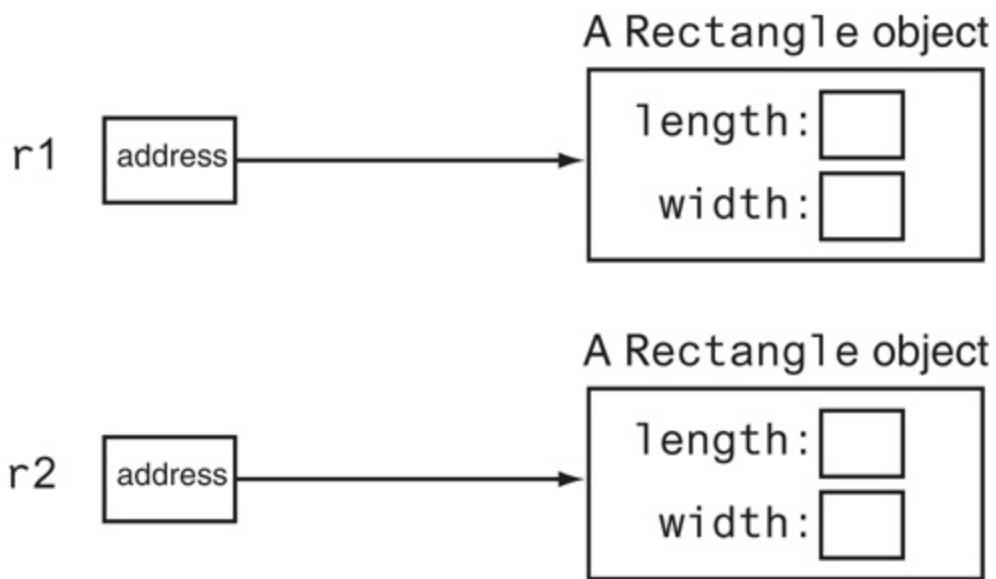
Checkpoint

1. 3.11 Assume that `r1` and `r2` are variables that reference `Rectangle` objects, and the following statements are executed:

```
r1.setLength(5.0);
r2.setLength(10.0);
r1.setWidth(20.0);
r2.setWidth(15.0);
```

Fill in the boxes in [Figure 3-19](#) that represent each object's `length` and `width` fields.

Figure 3-19 Fill in the boxes for each field



[Figure 3-19 Full Alternative Text](#)

3.4 Constructors

Concept:

A constructor is a method that is automatically called when an object is created.

A *constructor* is a method that is automatically called when an instance of a class is created. Constructors normally perform initialization or setup operations, such as storing initial values in instance fields.



[VideoNote](#) Initializing an Object with a Constructor

A constructor method has the same name as the class. For example, [Code Listing 3-10](#) shows the first few lines of a new version of the Rectangle class. In this version of the class, a constructor has been added. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03\Rectangle Class Phase 6*.)

Code Listing 3-10 (Rectangle.java)

```
1 /**
2  * Rectangle class, Phase 6
3  */
4
5 public class Rectangle
6 {
7     private double length;
8     private double width;
9
10    /**
```

```
11  * Constructor
12  */
13
14  public Rectangle(double len, double w)
15  {
16      length = len;
17      width = w;
18  }
... The remainder of the class has not changed, and is not shown.
```

This constructor accepts two arguments, which are passed into the `len` and `w` parameter variables. The parameter variables are then assigned to the `length` and `width` fields.

Notice that the constructor's header doesn't specify a return type—not even `void`. This is because constructors are not executed by explicit function calls and cannot return a value. The method header for a constructor takes the following general form:

AccessSpecifier ClassName(Arguments...)

Here is an example statement that declares the variable `box`, creates a `Rectangle` object, and passes the values 7.0 and 14.0 to the constructor.

```
Rectangle box = new Rectangle(7.0, 14.0);
```

After this statement executes, `box` will reference a `Rectangle` object whose `length` field is set to 7.0, and whose `width` field is set to 14.0. The program in [Code Listing 3-11](#) demonstrates the `Rectangle` class constructor. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03\Rectangle Class Phase 6.*)

Code Listing 3-11 (ConstructorDemo.java)

```
1 /**
2  * This program demonstrates the Rectangle class's
3  * constructor.
```

```
4  */
5
6 public class ConstructorDemo
7 {
8     public static void main(String[] args)
9     {
10     Rectangle box = new Rectangle(5.0, 15.0);
11
12     System.out.println("The box's length is " +
13         box.getLength());
14     System.out.println("The box's width is " +
15         box.getWidth());
16     System.out.println("The box's area is " +
17         box.getArea());
18 }
19 }
```

Program Output

```
The box's length is 5.0
The box's width is 15.0
The box's area is 75.0
```

The program in [Code Listing 3-11](#) uses the new key word to create a Rectangle object as part of the box variable's declaration statement. Recall that the new key word can be used in a simple assignment statement as well. For example, the following statement can be used to declare box as a Rectangle variable:

```
Rectangle box;
```

Then, the following statement can be used to create a Rectangle object and pass the values 7.0 and 14.0 to its constructor.

```
box = new Rectangle(7.0, 14.0);
```

The *RoomConstructor.java* program in [Code Listing 3-12](#) uses this technique. It is a modification of the *RoomAreas.java* program presented earlier in this chapter. (If you have downloaded the book's source code from www.pearsonhighered.com/gaddis, you will find this file in the folder *Chapter 03\Rectangle Class Phase 6.*)

Code Listing 3-12

(RoomConstructor.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program creates three instances of the Rectangle
5  * class and passes arguments to the constructor.
6  */
7
8 public class RoomConstructor
9 {
10    public static void main(String [] args)
11    {
12        double roomLength, // To hold a room's length
13            roomWidth, // To hold a room's width
14            totalArea; // To hold the total area
15
16        // Declare Rectangle variables to reference
17        // objects for the kitchen, bedroom, and den.
18        Rectangle kitchen, bedroom, den;
19
20        // Create a Scanner object for keyboard input.
21        Scanner keyboard = new Scanner(System.in);
22
23        // Get and store the dimensions of the kitchen.
24        System.out.print("What is the kitchen's length? ");
25        roomLength = keyboard.nextDouble();
26        System.out.print("What is the kitchen's width? ");
27        roomWidth = keyboard.nextDouble();
28        kitchen = new Rectangle(roomLength, roomWidth);
29
30        // Get and store the dimensions of the bedroom.
31        System.out.print("What is the bedroom's length? ");
32        roomLength = keyboard.nextDouble();
33        System.out.print("What is the bedroom's width? ");
34        roomWidth = keyboard.nextDouble();
35        bedroom = new Rectangle(roomLength, roomWidth);
36
37        // Get and store the dimensions of the den.
38        System.out.print("What is the den's length? ");
39        roomLength = keyboard.nextDouble();
40        System.out.print("What is the den's width? ");
41        roomWidth = keyboard.nextDouble();
```

```

42     den = new Rectangle(roomLength, roomWidth);
43
44     // Calculate the total area of the rooms.
45     totalArea = kitchen.getArea() + bedroom.getArea() +
46             den.getArea();
47
48     // Display the total area of the rooms.
49     System.out.println("The total area of the rooms is " +
50                         totalArea);
51 }
52 }
```

Program Output with Example Input Shown in Bold

What is the kitchen's length? **10** 
 What is the kitchen's width? **14** 
 What is the bedroom's length? **15** 
 What is the bedroom's width? **12** 
 What is the den's length? **20** 
 What is the den's width? **30** 
 The total area of the rooms is 920.0

In the program, the following statement in line 18 declares kitchen, bedroom, and den as Rectangle variables:

```
Rectangle kitchen, bedroom, den;
```

These variables do not yet reference instances of the Rectangle class, however. Because these variables do not yet hold an object's address, they are *uninitialized reference variables*. These variables cannot be used until they reference objects. The following statement, which appears in line 28, creates a Rectangle object, passes the roomLength and roomWidth variables as arguments to the constructor, and assigns the object's address to the kitchen variable.

```
kitchen = new Rectangle(roomLength, roomWidth);
```

After this statement executes, the kitchen variable will reference the Rectangle object. Similar statements also appear later in the program that cause the bedroom and den variables to reference objects.

The Default Constructor

When an object is created, its constructor is always called. But what if we do not write a constructor in the object's class? If you do not write a constructor in a class, Java automatically provides one when the class is compiled. The constructor that Java provides is known as the default constructor. The default constructor doesn't accept arguments. It sets all of the class's numeric fields to 0, boolean fields to false, and char fields to the Unicode value 0. If the object has any fields that are reference variables, the default constructor sets them to the special value `null`, which means that they do not reference anything.

The *only* time that Java provides a default constructor is when you do not write your own constructor for a class. For example, at the beginning of this chapter, we developed the `Rectangle` class without writing a constructor for it. When we compiled the class, the compiler generated a default constructor that set both the `length` and `width` fields to 0.0. Assume that the following code uses that version of the class to create a `Rectangle` object:

```
// We wrote no constructor for the Rectangle class.  
Rectangle r = new Rectangle(); // Calls the default constructor
```

When we created `Rectangle` objects using that version of the class, we did not pass any arguments to the default constructor, because the default constructor doesn't accept arguments.

Later, we added our own constructor to the class. The constructor we added accepts arguments for the `length` and `width` fields. When we compiled the class at that point, Java did not provide a default constructor. The constructor that we added became the only constructor the class had. When we create `Rectangle` objects with that version of the class, we *must* pass the `length` and `width` arguments to the constructor. Using that version of the class, the following statement would cause an error because we have not provided arguments for the constructor.

```
// Now we wrote our own constructor for the Rectangle class.  
Rectangle box = new Rectangle(); // Error! Must now pass argument
```

Because we have added our own constructor, which requires two arguments, the class no longer has a default constructor.

No-Arg Constructors

A constructor that does not accept arguments is known as a *no-arg constructor*. The default constructor doesn't accept arguments, so it is considered a no-arg constructor. In addition, you can write your own no-arg constructor. For example, suppose we wrote the following constructor for the Rectangle class:

```
public Rectangle()
{
    length = 1.0;
    width = 1.0;
}
```

If we were using this constructor in our Rectangle class, we would not pass any arguments when creating a Rectangle object. The following code shows an example. After this code executes, the Rectangle object's length and width fields would both be set to 1.0.

```
// Now we have written our own no-arg constructor.
Rectangle r = new Rectangle(); // Calls the no-arg constructor
```

Showing Constructors in a UML Diagram

There is more than one accepted way of showing a class's constructor in a UML diagram. In this book, we simply show a constructor just as any other method, except we list no return type. [Figure 3-20](#) shows a UML diagram for the Rectangle class with the constructor listed.

Figure 3-20 UML diagram for

the Rectangle class showing the constructor

Rectangle
- length : double - width : double
+ Rectangle(len : double, w : double) + setLength(len : double) : void + setWidth(w : double) : void + getLength() : double + getWidth() : double + getArea() : double

[Figure 3-20 Full Alternative Text](#)

The String Class Constructor

You create primitive variables with simple declaration statements, and you create objects with the new operator. There is one class, however, that can be instantiated without the new operator: the String class. Because string operations are so common, Java allows you to create String objects in the same way that you create primitive variables. Here is an example:

```
String name = "Joe Mahoney";
```

This statement creates a String object in memory, initialized with the string literal "Joe Mahoney". The object is referenced by the name variable. If you wish, you can use the new operator to create a String object, and initialize the object by passing a string literal to the constructor, as shown here:

```
String name = new String("Joe Mahoney");
```



Note:

String objects are a special case in Java. Because they are so commonly used, Java provides numerous shortcut operations with String objects that are not possible with objects of other types. In addition to creating a String object without using the new operator, you can use the = operator to assign values to String objects, you can use the + operator to concatenate strings, and so forth. [Chapter 9](#) will discuss several of the String class methods.



Checkpoint

1. 3.12 How is a constructor named?
2. 3.13 What is a constructor's return type?
3. 3.14 Assume that the following is a constructor, which appears in a class:

```
classAct(int number)
{
    item = number;
}
```

1. What is the name of the class that this constructor appears in?
2. Write a statement that creates an object from the class and passes the value 25 as an argument to the constructor.

3.5 A BankAccount Class

The Rectangle class discussed in the previous section allows you to create objects that describe rectangles. Now we will look at a class that is modeled after a more tangible object: a bank account. Objects that are created from this class will simulate bank accounts, allowing us to perform operations such as making deposits, making withdrawals, calculating and adding interest, and getting the current balance. A UML diagram for the BankAccount class is shown in [Figure 3-21](#).

Figure 3-21 UML diagram for the BankAccount class

BankAccount
- balance : double - interestRate : double - interest : double
+ BankAccount(startBalance : double, intRate : double) + deposit(amount : double) : void + withdraw(amount : double) : void + addInterest() : void + getBalance() : double + getInterest() : double

[Figure 3-21 Full Alternative Text](#)

Here is a summary of the BankAccount class's fields:

- `balance` is a `double` that holds an account's current balance.
- `interestRate` is a `double` that holds the monthly interest rate for an account. The monthly interest rate is the annual interest rate divided by 12. For example, if the annual interest rate is 3 percent, then the monthly interest rate is 0.25 percent. We would store this value as 0.0025 in the `interestRate` field.
- `interest` is a `double` that holds the amount of interest earned for an account.

Here is a summary of the class's methods:

- The constructor has two parameter variables: `startBalance` and `intRate`. Both parameters are `doubles`. When the constructor executes, the account's starting balance is passed into the `startBalance` parameter, and the account's monthly interest rate is passed into the `intRate` parameter. The constructor assigns `startBalance` to the `balance` field and `intRate` to the `interestRate` field. Additionally, the constructor assigns 0.0 to the `interest` field.
- The `deposit` method has a parameter, `amount`, that is a `double`. When the method is called, an amount that is to be deposited into the account is passed into this parameter. The value of the parameter is then added to the value in the `balance` field.
- The `withdraw` method has a parameter, `amount`, that is a `double`. When the method is called, an amount that is to be withdrawn from the account is passed into this parameter. The value of the parameter is then subtracted from the value in the `balance` field.
- The `addInterest` method multiplies the `interestRate` field by the `balance` field to determine the monthly interest for the account. The amount of interest is assigned to the `interest` field and added to the value in the `balance` field.
- The `getBalance` method returns the value in the `balance` field, which is the current account balance.

- The `getInterest` method returns the value in the `interest` field, which is the amount of interest earned the last time the `addInterest` method was called.

[Code Listing 3-13](#) shows the code for the `BankAccount` class, which is stored in the `BankAccount.java` file.

Code Listing 3-13 (`BankAccount.java`)

```

1 /**
2  * BankAccount class
3  * This class simulates a bank account.
4 */
5
6 public class BankAccount
7 {
8     private double balance;    // Account balance
9     private double interestRate; // Interest rate
10    private double interest;   // Interest earned
11
12 /**
13  * The constructor initializes the balance
14  * and interestRate fields with the values
15  * passed to startBalance and intRate. The
16  * interest field is assigned 0.0.
17 */
18
19 public BankAccount(double startBalance,
20                     double intRate)
21 {
22     balance = startBalance;
23     interestRate = intRate;
24     interest = 0.0;
25 }
26
27 /**
28  * The deposit method adds the parameter
29  * amount to the balance field.
30 */
31
32 public void deposit(double amount)

```

```
33  {
34      balance += amount;
35  }
36
37 /**
38 * The withdraw method subtracts the
39 * parameter amount from the balance
40 * field.
41 */
42
43 public void withdraw(double amount)
44 {
45     balance -= amount;
46 }
47
48 /**
49 * The addInterest method adds the
50 * interest for the month to the balance field.
51 */
52
53 public void addInterest()
54 {
55     interest = balance * interestRate;
56     balance += interest;
57 }
58
59 /**
60 * The getBalance method returns the
61 * value in the balance field.
62 */
63
64 public double getBalance()
65 {
66     return balance;
67 }
68
69 /**
70 * The getInterest method returns the
71 * value in the interest field.
72 */
73
74 public double getInterest()
75 {
76     return interest;
77 }
78 }
```

The *AccountTest.java* program, shown in [Code Listing 3-14](#), demonstrates the BankAccount class.

Code Listing 3-14 (**AccountTest.java**)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the BankAccount class.
5 */
6
7 public class AccountTest
8 {
9     public static void main(String[] args)
10    {
11        BankAccount account; // To reference a BankAccount object
12        double balance, // The account's starting balance
13            interestRate, // The monthly interest rate
14            pay, // The user's pay
15            cashNeeded; // The amount of cash to withdraw
16
17        // Create a Scanner object for keyboard input.
18        Scanner keyboard = new Scanner(System.in);
19
20        // Get the starting balance.
21        System.out.print("What is your account's " +
22                        "starting balance? ");
23        balance = keyboard.nextDouble();
24
25        // Get the monthly interest rate.
26        System.out.print("What is your monthly interest rate? ");
27        interestRate = keyboard.nextDouble();
28
29        // Create a BankAccount object.
30        account = new BankAccount(balance, interestRate);
31
32        // Get the amount of pay for the month.
33        System.out.print("How much were you paid this month? ");
34        pay = keyboard.nextDouble();
35
36        // Deposit the user's pay into the account.
37        System.out.println("We will deposit your pay " +
```

```

38                     "into your account.");
39         account.deposit(pay);
40         System.out.println("Your current balance is $" +
41                         account.getBalance());
42
43     // Withdraw some cash from the account.
44     System.out.print("How much would you like " +
45                       "to withdraw? ");
46     cashNeeded = keyboard.nextDouble();
47     account.withdraw(cashNeeded);
48
49     // Add the monthly interest to the account.
50     account.addInterest();
51
52     // Display the interest earned and the balance.
53     System.out.println("This month you have earned $" +
54                         account.getInterest() +
55                         " in interest.");
56     System.out.println("Now your balance is $" +
57                         account.getBalance());
58 }
59 }
```

Program Output with Example Input Shown in Bold

What is your account's starting balance? **500** 
 What is your monthly interest rate? **0.045** 
 How much were you paid this month? **1000** 
 We will deposit your pay into your account.
 Your current balance is \$1500.0
 How much would you like to withdraw? **900** 
 This month you have earned \$27.0 in interest.
 Now your balance is \$627.0

Let's look at some of the details of this program. First, some variables are declared with the following statements, which appear in lines 11 through 15:

```

BankAccount account; // To reference a BankAccount object
double balance, // The account's starting balance
       interestRate, // The monthly interest rate
       pay, // The user's pay
       cashNeeded; // The amount of cash to withdraw
```

The first variable declared is `account`. This is a variable that will be used

later in the program to reference a BankAccount object. Note that the new key word is not used in this statement, so the variable does not yet reference an object. Then, the variables balance, interestRate, pay, and cashNeeded are declared. These will hold values that are input by the user.

Next, the following code appears in lines 20 through 27:

```
// Get the starting balance.  
System.out.print("What is your account's " +  
    "starting balance? ");  
balance = keyboard.nextDouble();  
  
// Get the monthly interest rate.  
System.out.print("What is your monthly interest rate? ");  
interestRate = keyboard.nextDouble();
```

This code asks the user to enter his or her account's starting balance and the monthly interest rate. These values are stored in the balance and interestRate variables. Then, the following code appears in lines 29 and 30:

```
// Create a BankAccount object.  
account = new BankAccount(balance, interestRate);
```

The statement shown in line 30 uses the new key word to create a BankAccount object. The balance and interestRate variables are passed as arguments. Next, the following code appears in lines 32 through 41:

```
// Get the amount of pay for the month.  
System.out.print("How much were you paid this month? ");  
pay = keyboard.nextDouble();  
  
// Deposit the user's pay into the account.  
System.out.println("We will deposit your pay " +  
    "into your account.");  
account.deposit(pay);  
System.out.println("Your current balance is $" +  
    account.getBalance());
```

First, this code asks the user to enter his or her pay for the month. The input is stored in the pay variable. The account object's deposit method is called, and the pay variable is passed as an argument. Here is the code for the deposit method, which appears in the BankAccount class in lines 32 through

35:

```
public void deposit(double amount)
{
    balance += amount;
}
```

The value in `pay` is passed into the `amount` parameter variable. Then, the statement in this method uses a combined assignment operator to add `amount` to the value already in `balance`. This statement is the same as:

```
balance = balance + amount;
```

Back in the `AccountTest` program, the `account` object's `getBalance` method is then called in line 41 to get the current balance, which is displayed on the screen. Next, the following code appears in lines 43 through 47:

```
// Withdraw some cash from the account.
System.out.print("How much would you like " +
                  "to withdraw? ");
cashNeeded = keyboard.nextDouble();
account.withdraw(cashNeeded);
```

This code asks the user for the amount to withdraw from the account, and the input is stored in the `cashNeeded` variable. This variable is then passed as an argument to the `account` object's `withdraw` method. Here is the code for the `withdraw` method, which appears in the `BankAccount` class, in lines 43 through 46:

```
public void withdraw(double amount)
{
    balance -= amount;
}
```

The value in `cashNeeded` is passed into the `amount` parameter variable. Then, the statement in line 45 uses a combined assignment operator to subtract `amount` from the value already in `balance`. This statement is the same as:

```
balance = balance - amount;
```

Back in the `AccountTest` program, the following code appears in lines 49 through 57:

```

// Add the monthly interest to the account.
account.addInterest();

// Display the interest earned and the balance.
System.out.println("This month you have earned $" +
    account.getInterest() +
    " in interest.");
System.out.println("Now your balance is $" +
    account.getBalance());

```

Line 50 calls the `account` object's `addInterest` method, which calculates an amount of interest, assigns that amount to the object's `interest` field, and adds that amount to the object's `balance` field. Then, the object's `getInterest` and `getBalance` methods are called to get the amount of interest and the current balance, which are displayed.

In the Spotlight: Creating the CellPhone Class



Wireless Solutions, Inc. is a business that sells cell phones and wireless service. You are a programmer in the company's IT department, and your team is designing a program to manage all of the cell phones that are in inventory. You have been asked to design a class that represents a cell phone. The data that should be kept as fields in the class are as follows:

- The name of the phone's manufacturer is assigned to the `manufacturer` field.
- The phone's model number is assigned to the `model` field.
- The phone's retail price is assigned to the `retailPrice` field.

The class will also have the following methods:

- A constructor that accepts arguments for the manufacturer, model number, and retail price.

- A `setManufact` method that accepts an argument for the manufacturer. This method allows us to change the value of the `manufact` field after the object has been created, if necessary.
- A `setModel` method that accepts an argument for the model. This method allows us to change the value of the `model` field after the object has been created, if necessary.
- A `setRetailPrice` method that accepts an argument for the retail price. This method allows us to change the value of the `retailPrice` field after the object has been created, if necessary.
- A `getManufact` method that returns the phone's manufacturer.
- A `getModel` method that returns the phone's model number.
- A `getRetailPrice` method that returns the phone's retail price.

[Figure 3-22](#) shows a UML diagram for the class. [Code Listing 3-15](#) shows the class definition.

Figure 3-22 UML diagram for the CellPhone class

CellPhone
<ul style="list-style-type: none"> - <code>manufact</code> : String - <code>model</code> : String - <code>retailPrice</code> : double
<ul style="list-style-type: none"> + <code>CellPhone(man : String, mod : String, price : double);</code> + <code>setManufact(man : String) : void</code> + <code>setModel(mod : String) : void</code> + <code>setRetailPrice(price : double) : void</code> + <code>getManufact() : String</code> + <code>getModel() : String</code> + <code>getRetailPrice() : double</code>

[Figure 3-22 Full Alternative Text](#)

Code Listing 3-15 (CellPhone.java)

```

1 /**
2  * The CellPhone class represents a cell phone.
3 */
4
5 public class CellPhone
6 {
7     // Fields
8     private String manufact; // Manufacturer
9     private String model;   // Model
10    private double retailPrice; // Retail price
11
12    /**
13     * The constructor accepts arguments for
14     * the phone's manufacturer, model number,
15     * and retail price.
16    */
17
18    public CellPhone(String man, String mod, double price)
19    {
20        manufact = man;
21        model = mod;

```

```
22     retailPrice = price;
23 }
24
25 /**
26 * The setManufact method accepts an argument for
27 * the phone's manufacturer name.
28 */
29
30 public void setManufact(String man)
31 {
32     manufact = man;
33 }
34
35 /**
36 * The setModelNumber method accepts an argument
37 * for the phone's model number.
38 */
39
40 public void setMod(String mod)
41 {
42     model = mod;
43 }
44
45 /**
46 * The setRetailPrice method accepts an argument
47 * for the phone's retail price.
48 */
49
50 public void setRetailPrice(double price)
51 {
52     retailPrice = price;
53 }
54
55 /**
56 * The getManufact method returns the name of
57 * the phone's manufacturer.
58 */
59
60 public String getManufact()
61 {
62     return manufact;
63 }
64
65 /**
66 * The getModel method returns the phone's
67 * model number.
68 */
69
```

```
70 public String getModel()
71 {
72     return model;
73 }
74
75 /**
76 * The getRetailPrice method returns the
77 * phone's retail price.
78 */
79
80 public double getRetailPrice()
81 {
82     return retailPrice;
83 }
84 }
```

The CellPhone class will be used by several programs that your team is developing. To perform a simple test of the class, you write the program shown in [Code Listing 3-16](#). This is a simple program that prompts the user for the phone's manufacturer, model number, and retail price. An instance of the CellPhone class is created, and the data is assigned to its attributes.

Code Listing 3-16 (CellPhoneTest.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program runs a simple test
5  * of the CellPhone class.
6 */
7
8 public class CellPhoneTest
9 {
10    public static void main(String[] args)
11    {
12        String testMan; // To hold a manufacturer
13        String testMod; // To hold a model number
14        double testPrice; // To hold a price
15
16        // Create a Scanner object for keyboard input.
17        Scanner keyboard = new Scanner(System.in);
```

```
18
19 // Get the manufacturer name.
20 System.out.print("Enter the manufacturer: ");
21 testMan = keyboard.nextLine();
22
23 // Get the model number.
24 System.out.print("Enter the model number: ");
25 testMod = keyboard.nextLine();
26
27 // Get the retail price.
28 System.out.print("Enter the retail price: ");
29 testPrice = keyboard.nextDouble();
30
31 // Create an instance of the CellPhone class,
32 // passing the data that was entered as arguments
33 // to the constructor.
34 CellPhone phone = new CellPhone(testMan, testMod, testPrice
35
36 // Get the data from the phone and display it.
37 System.out.println();
38 System.out.println("Here is the data that you provided:");
39 System.out.println("Manufacturer: " + phone.getManufact());
40 System.out.println("Model number: " + phone.getModel());
41 System.out.println("Retail price: " + phone.getRetailPrice(
42 }
43 }
```

Program Output with Example Input Shown in Bold

Enter the manufacturer: Acme Electronics 
Enter the model number: M1000 
Enter the retail price: 299.99 

Here is the data that you provided:
Manufacturer: Acme Electronics
Model Number: M1000
Retail Price: \$299.99

3.6 Classes, Variables, and Scope

Concept:

Instance fields are visible to all of the class's instance methods. Local variables, including parameter variables, are visible only to statements in the method where they are declared.

Recall from [Chapter 2](#) that a variable's scope is the part of a program where the variable can be accessed by its name. A variable's name is visible only to statements inside the variable's scope. The location of a variable's declaration determines the variable's scope. So far, you have seen variables declared in the following locations:

- Inside a method. Variables declared inside a method are known as local variables.
- Inside a class, but not inside any method. Variables that are declared inside a class, but not inside any method are known as fields.
- Inside the parentheses of a method header. Variables that are declared inside the parentheses of a method header are known as parameter variables.

The following list summarizes the scope for each of these types of variables.

- Local variables. A local variable's scope is the method in which it is declared, from the variable's declaration to the end of the method. Only statements in this area can access the variable.
- Fields. For now, we will define a field's scope as the entire class in which it is declared. A field can be accessed by the methods that are

members of the same class as the field. (To be completely accurate, an instance field may be accessed by any instance method that is a member of the same class. In [Chapter 7](#), we will discuss non-instance class members.)

- Parameter variables. A parameter variable's scope is the method in which it is declared. Only statements inside the method can access the parameter variable.

Shadowing

In [Chapter 2](#), you saw that you cannot have two local variables with the same name in the same scope. This applies to parameter variables as well. A parameter variable is, in essence, a local variable. So, you cannot give a parameter variable and a local variable in the same method the same name.

However, you can have a local variable or a parameter variable with the same name as a field. When you do, the name of the local or parameter variable *shadows* the name of the field. This means that the field name is hidden by the name of the local or parameter variable.

For example, assume that the `Rectangle` class's `setLength` method had been written in the following manner:

```
public void setLength(double len)
{
    int length;          // Local variable
    length = len;
}
```

In this code, a local variable is given the same name as the field. Therefore, the local variable's name shadows the field's name. When the statement `length = len;` is executed, the value of `len` is assigned to the local variable `length`, not to the field. The unintentional shadowing of field names can cause elusive bugs, so you need to be careful not to give local variables the same names as fields.

3.7 Packages and import Statements

Concept:

The classes in the Java API are organized into packages. An import statement tells the compiler in which package a class is located.

In [Chapter 2](#), you were introduced to the Java API, which is a standard library of prewritten classes. Each class in the Java API is designed for a specific purpose, and you can use the classes in your own programs. You've already used a few classes from the API, such as the `String` class, the `Scanner` class, and the `JOptionPane` class.

All of the classes in the Java API are organized into packages. A *package* is simply a group of related classes. Each package also has a name. For example, the `Scanner` class is in the `java.util` package.

Many of the classes in the Java API are not automatically available to your program. Quite often, you have to *import* an API class in order to use it. You use the `import` key word to import a class. For example, the following statement is required to import the `Scanner` class:

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

This statement tells the compiler that the `Scanner` class is located in the `java.util` package. Without this statement, the compiler will not be able to locate the `Scanner` class, and the program will not compile.

Explicit and Wildcard import Statements

There are two types of `import` statements: explicit and wildcard. An *explicit import* statement identifies the package location of a single class. For example, the following statement explicitly identifies the location of the `Scanner` class:

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

The `java.util` package has several other classes in it as well as the `Scanner` class. For example, in [Chapter 7](#), we will study the `ArrayList` class, and in [Chapter 8](#), we will study the `StringTokenizer` class. Both of these classes are part of the `java.util` package. If a program needs to use the `Scanner` class, the `ArrayList` class, and the `StringTokenizer` class, it will have to import all three of these classes. One way to do this is to write explicit `import` statements for each class, as shown here:

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.StringTokenizer;
```

Another way to import all of these classes is to use a wildcard `import` statement. A *wildcard import* statement tells the compiler to import all of the classes in a package. Here is an example:

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

The `.*` that follows the package name tells the compiler to import all the classes that are part of the `java.util` package. Using a wildcard `import` statement does not affect the performance or the size of your program. It merely tells the compiler that you want to make every class in a particular package available to your program.

The `java.lang` Package

The Java API has one package, `java.lang`, that is automatically imported into every Java program. This package contains general classes, such as `String` and `System`, that are fundamental to the Java programming language. You do not have to write an `import` statement for any class that is part of the `java.lang` package.

Other API Packages

There are numerous packages in the Java API. [Table 3-2](#) lists a few of them.

Table 3-2 A few of the standard Java packages

Package	Description
java.io	Provides classes that perform various types of input and output.
java.lang	Provides general classes for the Java language. This package is automatically imported.
java.net	Provides classes for network communications.
java.security	Provides classes that implement security features.
java.sql	Provides classes for accessing databases using structured query language.
java.text	Provides various classes for formatting text.
java.util	Provides various utility classes.
javafx	Provides classes for creating graphical user interfaces.

For more details about packages, see Appendix G, available on this book's online resource page at www.pearsonhighered.com/gaddis.

3.8 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities

Concept:

One of the first steps in creating an object-oriented application is determining the classes that are necessary, and their responsibilities within the application.

So far you have learned the basics of writing a class, creating an object from the class, and using the object to perform operations. Although this knowledge is necessary to create an object-oriented application, it is not the first step. The first step is to analyze the problem that you are trying to solve, and determine the classes that you will need. In this section, we will discuss a simple technique for finding the classes in a problem and determining their responsibilities.

Finding the Classes

When developing an object-oriented application, one of your first tasks is to identify the classes that you will need to create. Typically, your goal is to identify the different types of real-world objects that are present in the problem, then create classes for those types of objects within your application.

Over the years, software professionals have developed numerous techniques for finding the classes in a given problem. One simple and popular technique involves the following steps.

1. Get a written description of the problem domain.
2. Identify all the nouns (including pronouns and noun phrases) in the description. Each of these is a potential class.
3. Refine the list to include only the classes that are relevant to the problem.

Let's take a closer look at each of these steps.

Writing a Description of the Problem Domain

The *problem domain* is the set of real-world objects, parties, and major events related to the problem. If you adequately understand the nature of the problem you are trying to solve, you can write a description of the problem domain yourself. If you do not thoroughly understand the nature of the problem, you should have an expert write the description for you.

For example, suppose we are programming an application that the manager of Joe's Automotive Shop will use to print service quotes for customers. Here is a description that an expert, perhaps Joe himself, might have written:

Joe's Automotive Shop services foreign cars and specializes in servicing cars made by Mercedes, Porsche, and BMW. When a customer brings a car to the shop, the manager gets the customer's name, address, and telephone number. The manager then determines the make, model, and year of the car and gives the customer a service quote. The service quote shows the estimated parts charges, estimated labor charges, sales tax, and total estimated charges.

The problem domain description should include any of the following:

- Physical objects, such as vehicles, machines, or products
- Any role played by a person, such as manager, employee, customer,

teacher, student, and so on.

- The results of a business event, such as a customer order, or in this case a service quote
- Recordkeeping items, such as customer histories and payroll records

Identify All of the Nouns

The next step is to identify all of the nouns and noun phrases. (If the description contains pronouns, include them too.) Here's another look at the previous problem domain description. This time the nouns and noun phrases appear in bold.

Joe's Automotive Shop services **foreign cars** and specializes in servicing **cars** made by **Mercedes**, **Porsche**, and **BMW**. When a **customer** brings a **car** to the **shop**, the **manager** gets the **customer's name, address, and telephone number**. The **manager** then determines the **make, model, and year** of the **car** and gives the **customer** a **service quote**. The **service quote** shows the **estimated parts charges, estimated labor charges, sales tax, and total estimated charges**.

Notice that some of the nouns are repeated. The following list shows all of the nouns without duplicating any of them.

- address
- BMW
- car
- cars
- customer
- estimated labor charges

- estimated parts charges
- foreign cars
- Joe's Automotive Shop
- make
- manager
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

Refining the List of Nouns

The nouns that appear in the problem description are merely candidates to become classes. It might not be necessary to make classes for them all. The next step is to refine the list to include only the classes that are necessary to solve the particular problem at hand. We will look at the common reasons that a noun can be eliminated from the list of potential classes.

1. Some of the nouns really mean the same thing.

In this example, the following sets of nouns refer to the same thing:

- **cars and foreign cars**

These both refer to the general concept of a car.

- **Joe's Automotive Shop and shop**

Both of these refer to the company “Joe’s Automotive Shop.”

We can settle on a single class for each of these. In this example, we will arbitrarily eliminate **foreign cars** from the list, and use the word **cars**.

Likewise we will eliminate **Joe’s Automotive Shop** from the list and use the word **shop**. The updated list of potential classes is:

- address
- BMW
- car
- cars
- customer
- estimated labor charges
- estimated parts charges
- **foreign cars**
- ~~Joe's Automotive Shop~~
- make
- manager

- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

Because cars and foreign cars mean the same thing in this problem, we have eliminated foreign cars. Also, because Joe's Automotive Shop and shop mean the same thing, we have eliminated Joe's Automotive Shop.

2. Some nouns might represent items that we do not need to be concerned with in order to solve the problem.

A quick review of the problem description reminds us of what our application should do: print a service quote. In this example, we can eliminate two unnecessary classes from the list:

- We can cross **shop** off the list because our application needs to be concerned only with individual service quotes. It doesn't need to work with or determine any company-wide information. If the problem description asked us to keep a total of all the service quotes, then it would make sense to have a class for the shop.
- We will not need a class for the **manager** because the problem statement does not direct us to process any information about the

manager. If there were multiple shop managers, and the problem description had asked us to record which manager generated each service quote, then it would make sense to have a class for the manager.

The updated list of potential classes at this point is:

- address
- BMW
- car
- cars
- customer
- estimated labor charges
- estimated parts charges
- ~~foreign cars~~
- ~~Joe's Automotive Shop~~
- make
- ~~manager~~
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote

- shop
- telephone number
- total estimated charges
- year

Our problem description does not direct us to process any information about the shop, or any information about the manager, so we have eliminated those from the list.

3. Some of the nouns might represent objects, not classes.

We can eliminate **Mercedes**, **Porsche**, and **BMW** as classes because, in this example, they all represent specific cars and can be considered instances of a **cars** class. Also, we can eliminate the word **car** from the list. In the description, it refers to a specific car brought to the shop by a customer. Therefore, it would also represent an instance of a **cars** class. At this point, the updated list of potential classes is:

- address
- **BMW**
- **car**
- cars
- customer
- estimated labor charges
- estimated parts charges
- **foreign cars**
- **Joe's Automotive Shop**

- manager
- make
- Mercedes
- model
- name
- Porsche
- sales tax
- service quote
- shop
- telephone number
- total estimated charges
- year

We have eliminated Mercedes, Porsche, BMW, and car because they are all instances of a cars class. That means that these nouns identify objects, not classes.

Tip:

Some object-oriented designers take note of whether a noun is plural or singular. Sometimes, a plural noun will indicate a class, and a singular noun will indicate an object.

4. **Some of the nouns might represent simple values that can be stored in a primitive variable and do not require a class.**

Remember, a class contains fields and methods. Fields are related items that are stored within an object of the class, and define the object's state. Methods are actions or behaviors that may be performed by an object of the class. If a noun represents a type of item that would not have any identifiable fields or methods, then it can probably be eliminated from the list. To help determine whether a noun represents an item that would have fields and methods, ask the following questions about it:

- Would you use a group of related values to represent the item's state?
- Are there any obvious actions to be performed by the item?

If the answers to both of these questions are no, then the noun probably represents a value that can be stored in a primitive variable. If we apply this test to each of the nouns that remain in our list, we can conclude that the following are probably not classes: **address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges, and year**. These are all simple string or numeric values that can be stored in primitive variables. Here is the updated list of potential classes:

- address
- BMW
- car
- cars
- customer
- ~~estimated labor charges~~
- ~~estimated parts charges~~
- ~~foreign cars~~
- ~~Joe's Automotive Shop~~

- `make`
- `manager`
- `Mercedes`
- `model`
- `name`
- `Porsche`
- `sales tax`
- `service quote`
- `shop`
- `telephone number`
- ~~total estimated charges~~
- `year`

We have eliminated address, estimated labor charges, estimated parts charges, make, model, name, sales tax, telephone number, total estimated charges, and year as classes because they represent simple values that can be stored in primitive variables.

As you can see from the list, we have eliminated everything except `cars`, `customer`, and `service quote`. This means that in our application, we will need classes to represent cars, customers, and service quotes. Ultimately, we will write a `Car` class, a `Customer` class, and a `ServiceQuote` class.

Identifying a Class's Responsibilities

Once the classes have been identified, the next task is to identify each class's responsibilities. A class's *responsibilities* are:

- the information the class is responsible for knowing and;
- the actions the class is responsible for doing.

When you have identified the things that a class is responsible for knowing, then you have identified the class's attributes. These values will be stored in fields. Likewise, when you have identified the actions that a class is responsible for doing, you have identified its methods.

It is often helpful to ask the questions, "In the context of this problem, what must the class know? What must the class do?" The first place to look for the answers is in the description of the problem domain. Many of the things that a class must know and do will be mentioned. Some class responsibilities, however, might not be directly mentioned in the problem domain, so brainstorming is often required. Let's apply this methodology to the classes we previously identified from our problem domain.

The Customer class

In the context of our problem domain, what must the `Customer` class know? The description directly mentions the following items, which are all attributes of a customer:

- the customer's name
- the customer's address
- the customer's telephone number

These are all values that can be represented as strings and stored in the class's fields. The `Customer` class can potentially know many other things. One mistake that can be made at this point is to identify too many things that an object is responsible for knowing. In some applications, a `Customer` class might know the customer's email address. This particular problem domain

does not mention that the customer's email address is used for any purpose, so we should not include it as a responsibility.

Now let's identify the class's methods. In the context of our problem domain, what must the Customer class do? The only obvious actions are:

- create an object of the Customer class.
- set and get the customer's name.
- set and get the customer's address.
- set and get the customer's telephone number.

From this list we can see that the Customer class will have a constructor, as well as accessor and mutator methods, for each of its fields. [Figure 3-23](#) shows a UML diagram for the Customer class.

Figure 3-23 UML diagram for the Customer class

Customer
- name : String - address : String - phone : String
+ Customer() + setName(n : String) : void + setAddress(a : String) : void + setPhone(p : String) : void + getName() : String + getAddress() : String + getPhone() : String

[Figure 3-23 Full Alternative Text](#)

The Car Class

In the context of our problem domain, what must an object of the car class know? The following items are all attributes of a car, and are mentioned in the problem domain:

- the car's make
- the car's model
- the car's year

Now let's identify the class's methods. In the context of our problem domain, what must the car class do? Once again, the only obvious actions are the standard set of methods that we will find in most classes (constructors, accessors, and mutators). Specifically, the actions are:

- create an object of the car class.
- set and get the car's make.
- set and get the car's model.
- set and get the car's year.

[Figure 3-24](#) shows a UML diagram for the car class at this point.

Figure 3-24 UML diagram for the car class

Car
<ul style="list-style-type: none"> - make : String - model : String - year : int
<ul style="list-style-type: none"> + Car() + setMake(m : String) : void + setModel(m : String) : void + setYear(y : int) : void + getMake() : String + getModel() : String + getYear() : int

[Figure 3-24 Full Alternative Text](#)

The ServiceQuote Class

In the context of our problem domain, what must an object of the ServiceQuote class know? The problem domain mentions the following items:

- the estimated parts charges
- the estimated labor charges
- the sales tax
- the total estimated charges

Careful thought and a little brainstorming will reveal that two of these items are the results of calculations: sales tax and total estimated charges. These items are dependent on the values of the estimated parts and labor charges. In order to avoid the risk of holding stale data, we will not store these values in fields. Rather, we will provide methods that calculate these values and return them. The other methods that we will need for this class are a constructor and the accessors and mutators for the estimated parts charges and estimated

labor charges fields. [Figure 3-25](#) shows a UML diagram for the ServiceQuote class.

Figure 3-25 UML diagram for the ServiceQuote class

ServiceQuote
- partsCharges : double - laborCharges : double
+ ServiceQuote() + setPartsCharges(c : double) : void + setLaborCharges(c : double) : void + getPartsCharges() : double + getLaborCharges() : double + salesTax() : double + totalCharges() : double

[Figure 3-25 Full Alternative Text](#)

This Is Only the Beginning

You should look at the process that we have discussed in this section merely as a starting point. It's important to realize that designing an object-oriented application is an iterative process. It may take you several attempts to identify all of the classes that you will need and determine all of their responsibilities. As the design process unfolds, you will gain a deeper understanding of the problem. Consequently, you will see ways to improve the design.



Checkpoint

1. 3.15 What is a problem domain?
2. 3.16 When designing an object-oriented application, who should write a description of the problem domain?
3. 3.17 How do you identify the potential classes in a problem domain description?
4. 3.18 What are a class's responsibilities?
5. 3.19 What two questions should you ask to determine a class's responsibilities?
6. 3.20 Will all of a class's actions always be directly mentioned in the problem domain description?

3.9 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- Putting a semicolon at the end of a method header. A semicolon never appears at the end of a method header.
- Declaring a variable to reference an object, but forgetting to use the new key word to create the object. Declaring a variable to reference an object does not create an object. You must use the new key word to create the object.
- Forgetting the parentheses that must appear after the class name, which appears after the new key word. The name of a class appears after the new key word, and a set of parentheses appears after the class name. You must write the parentheses even if no arguments are passed to the constructor.
- Forgetting to provide arguments when a constructor requires them. When using a constructor that has parameters, you must provide arguments for them.
- Forgetting the parentheses in a method call. You must write the parentheses after the name of the method in a statement that calls the method, even if no arguments are passed to the method.
- Forgetting to pass arguments to methods that require them. If a method has parameters, you must provide arguments when calling the method.
- In a method, unintentionally declaring a local variable with the same name as a field of the same class. When a method's local variable has the same name as a field in the same class, the local variable's name shadows the field's name.
- Passing an argument to a method that is incompatible with the parameter

variable's data type. An argument that is passed to a method must be of a data type that is compatible with the parameter variable receiving it.

- Using a variable to receive a method's return value when the variable's data type is incompatible with the data type of the return value. A variable that receives a method's return value must be of a data type that is compatible with the data type of the return value.

Review Questions and Exercises

Multiple Choice and True/False

1. A collection of programming statements that specify the attributes and methods a particular type of object can have is .
 1. class
 2. method
 3. parameter
 4. instance
2. A class is analogous to a .
 1. blueprint
 2. house
 3. architect
 4. attribute
3. An object is a(n) .
 1. blueprint
 2. attribute
 3. variable
 4. instance

4. This is a member of a class that holds data.
 1. method
 2. instance
 3. field
 4. constructor
5. This key word causes an object to be created in memory.
 1. create
 2. new
 3. object
 4. construct
6. This key word causes a value to be sent back from a method to the statement that called it.
 1. send
 2. return
 3. value
 4. public
7. This is a method that gets a value from a class's field, but does not change it.
 1. accessor
 2. constructor
 3. void

4. mutator
8. This is a method that stores a value in a field or in some other way changes the value of a field.
 1. accessor
 2. constructor
 3. void
 4. mutator
9. When the value of an item is dependent on other data, and that item is not updated when the other data is changed, what has the value become?
 1. bitter
 2. stale
 3. asynchronous
 4. moldy
10. This is a method that is automatically called when an instance of a class is created.
 1. accessor
 2. constructor
 3. void
 4. mutator
11. When a local variable has the same name as a field, the local variable's name does what to the field's name?
 1. shadows

2. complements
 3. deletes
 4. merges with
12. If you do not write a constructor for a class, this is automatically provided for the class.
1. accessor method
 2. default instance
 3. default constructor
 4. predefined constructor
13. A class's responsibilities are .
1. the objects created from the class
 2. things the class knows
 3. actions the class performs
 4. Both b and c
14. **True or False:** The occurrence of a string literal in a Java program causes a `String` object to be created in memory, initialized with the string literal.
15. **True or False:** When passing an argument to a method, the argument's data type must be compatible with the parameter variable's data type.
16. **True or False:** When passing multiple arguments to a method, the order in which the arguments are passed is not important.
17. **True or False:** Each instance of a class has its own set of instance fields.

18. **True or False:** When you write a constructor for a class, it still has the default constructor that Java automatically provides.
19. **True or False:** To find the classes needed for an object-oriented application, you identify all of the verbs in a description of the problem domain.

Find the Error

1. Find the error in the following class:

```
public class MyClass
{
    private int x;
    private double y;

    public void MyClass(int a, double b)
    {
        x = a;
        y = b;
    }
}
```

2. Assume that the following method is a member of a class. Find the error.

```
public void total(int value1, value2, value3)
{
    return value1 + value2 + value3;
}
```

3. The following statement attempts to create a Rectangle object. Find the error.

```
Rectangle box = new Rectangle;
```

Algorithm Workbench

1. Design a class named Pet, which should have the following attributes:

- name . The name attribute holds the name of a pet.
- animal . The animal attribute holds the type of animal that a pet is. Example values are “Dog”, “Cat”, and “Bird”.
- age . The age attribute holds the pet’s age.

The Pet class should also have the following methods:

- setName . The setName method stores a value in the name attribute.
 - setAnimal . The setAnimal method stores a value in the animal attribute.
 - setAge . The setAge method stores a value in the age attribute.
 - getName . The getName method returns the value of the name attribute.
 - getAnimal . The getAnimal method returns the value of the animal attribute.
 - getAge . The getAge method returns the value of the age attribute.
1. Draw a UML diagram of the class. Be sure to include notation showing each attribute’s and method’s access specification and data type. Also include notation showing any method parameters and their data types.
 2. Write the Java code for the Pet class.
2. Look at the following partial class definition, then respond to the questions that follow it:

```
public class Book
{
    private String title;
    private String author;
    private String publisher;
    private int copiesSold;
```

}

1. Write a constructor for this class. The constructor should accept an argument for each of the fields.
2. Write accessor and mutator methods for each field.
3. Draw a UML diagram for the class, including the methods you have written.
3. Look at the following description of a problem domain:

The bank offers the following types of accounts to its customers: savings accounts, checking accounts, and money market accounts. Customers are allowed to deposit money into an account (thereby increasing its balance), withdraw money from an account (thereby decreasing its balance), and earn interest on an account. Each account has an interest rate.

Assume that you are writing an application that will calculate the amount of interest earned for a bank account.

1. Identify the potential classes in this problem domain.
2. Refine the list to include only the necessary class or classes for this problem.
3. Identify the responsibilities of the class or classes.

Short Answer

1. What is the difference between a class and an instance of the class?
2. A contractor uses a blueprint to build a set of identical houses. Are classes analogous to the blueprint or the houses?
3. What is an accessor method? What is a mutator method?

4. Is it a good idea to make fields private? Why or why not?
5. If a class has a private field, what has access to the field?
6. What is the purpose of the new key word?
7. Assume a program named *MailList.java* is stored in the *DataBase* folder on your hard drive. The program creates objects of the *Customer* and *Account* classes. Describe the steps that the compiler goes through in locating and compiling the *Customer* and *Account* classes.
8. Explain what is meant by the phrase “pass by value.”
9. Why are constructors useful for performing “start-up” operations?
10. What is the difference between a field and an attribute?
11. What is the difference between an argument and a parameter variable?
12. Under what circumstances does Java automatically provide a default constructor for a class?
13. What do you call a constructor that accepts no arguments?

Programming Challenges

1. Employee Class

Write a class named `Employee` that has the following fields:

- `name`. The `name` field is a `String` object that holds the employee’s name.
- `idNumber`. The `idNumber` is an `int` variable that holds the employee’s ID number.
- `department`. The `department` field is a `String` object that holds the name of the department where the employee works.
- `position`. The `position` field is a `String` object that holds the employee’s job title.

Write appropriate mutator methods that store values in these fields, and accessor methods that return the values in these fields. Once you have written the class, write a separate program that creates three `Employee` objects to hold the following data:

Name	ID Number	Department	Position
Susan Meyers	47899	Accounting	Vice President
Mark Jones	39119	IT	Programmer
Joy Rogers	81774	Manufacturing	Engineer

The program should store this data in the three objects, then display the data for each employee on the screen.

2. Car Class

Write a class named `Car` that has the following fields:

- `yearModel`. The `yearModel` field is an `int` that holds the car’s year model.
- `make`. The `make` field is a `String` object that holds the make of the car, such as “Ford”, “Chevrolet”, “Honda”, etc.
- `speed`. The `speed` field is an `int` that holds the car’s current speed.

In addition, the class should have the following methods.

- Constructor. The constructor should accept the car’s year model and make as arguments. These values should be assigned to the object’s `yearModel` and `make` fields. The constructor should also assign 0 to the `speed` field.
- Accessor. The appropriate accessor methods get the values stored in an object’s `yearModel`, `make`, and `speed` fields.
- `accelerate`. The `accelerate` method should add 5 to the `speed` field each time it is called.
- `brake`. The `brake` method should subtract 5 from the `speed` field each time it is called.

Demonstrate the class in a program that creates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then, call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

3. Personal Information Class

Design a class that holds the following personal data: name, address, age, and phone number. Write appropriate accessor and mutator methods. Demonstrate the class by writing a program that creates three instances of it. One instance should hold your information, and the other two should hold your friends’ or family members’ information.



VideoNote The Personal Information Class Problem

4. Temperature Class

Write a Temperature class that will hold a temperature in Fahrenheit and provide methods to get the temperature in Fahrenheit, Celsius, and Kelvin. The class should have the following field:

- `ftemp`. A double that holds a Fahrenheit temperature.

The class should have the following methods:

- Constructor. The constructor accepts a Fahrenheit temperature (as a double) and stores it in the `ftemp` field.
- `setFahrenheit`. The `setFahrenheit` method accepts a Fahrenheit temperature (as a double) and stores it in the `ftemp` field.
- `getFahrenheit`. Returns the value of the `ftemp` field, as a Fahrenheit temperature (no conversion required).
- `getCelsius`. Returns the value of the `ftemp` field converted to Celsius.
- `getKelvin`. Returns the value of the `ftemp` field converted to Kelvin.

Use the following formula to convert the Fahrenheit temperature to Celsius:

$$\text{Celsius} = (5/9) \times (\text{Fahrenheit} - 32)$$

Use the following formula to convert the Fahrenheit temperature to Kelvin:

$$\text{Kelvin} = ((5/9) \times (\text{Fahrenheit} - 32)) + 273$$

Demonstrate the Temperature class by writing a separate program that asks the user for a Fahrenheit temperature. The program should create an instance of the Temperature class, with the value entered by the user passed to the constructor. The program should then call the object's methods to display the temperature in Celsius and Kelvin.

5. RetailItem Class

Write a class named `RetailItem` that holds data about an item in a retail store. The class should have the following fields:

- `description`. The `description` field is a `String` object that holds a brief description of the item.
- `unitsOnHand`. The `unitsOnHand` field is an `int` variable that holds the number of units currently in inventory.
- `price`. The `price` field is a `double` that holds the item's retail price.

Write appropriate mutator methods that store values in these fields, and accessor methods that return the values in these fields. Once you have written the class, write a separate program that creates three `RetailItem` objects and stores the following data in them:

Description	Units On Hand	Price
Item #1 Jacket	12	59.95
Item #2 Designer Jeans	40	34.95
Item #3 Shirt	20	24.95

6. Payroll Class

Design a `Payroll` class that has fields for an employee's name, ID number, hourly pay rate, and number of hours worked. Write the appropriate accessor and mutator methods, and a constructor that accepts the employee's name and ID number as arguments. The class should also have a method that returns the employee's gross pay, which

is calculated as the number of hours worked multiplied by the hourly pay rate. Write a program that demonstrates the class by creating a `Payroll` object, then asks the user to enter the data for an employee. The program should display the amount gross pay earned.

7. Widget Factory

Design a class for a widget manufacturing plant. The class should have a method whose argument is the number of widgets that must be produced. The class should have another method that calculates how many days it will take to produce the number of widgets. (Assume 10 widgets can be produced each hour. The plant operates two shifts of eight hours each per day.)

Demonstrate the class by writing a separate program that creates an instance of the class. The program should pass a number of widgets to the object, and call the object's method that displays the number of days it will take to produce that many widgets.

8. TestScores Class

Design a `TestScores` class that has fields to hold three test scores. The class should have accessor and mutator methods for the test score fields, and a method that returns the average of the test scores. Demonstrate the class by writing a separate program that creates an instance of the class. The program should ask the user to enter three test scores, which are stored in the `TestScores` object. Then, the program should display the average of the scores, as reported by the `TestScores` object.

9. Circle Class

Write a `Circle` class that has the following fields:

- `radius`. a `double`
- `PI`. a `final double` initialized with the value 3.14159

The class should have the following methods:

- Constructor. Accepts the radius of the circle as an argument.
- `setRadius`. A mutator method for the radius field.
- `getRadius`. An accessor method for the radius field.
- `getArea`. Returns the area of the circle, which is calculated as

$$\text{area} = \text{PI} * \text{radius} * \text{radius}$$
- `getDiameter`. Returns the diameter of the circle, which is calculated as

$$\text{diameter} = \text{radius} * 2$$
- `getCircumference`. Returns the circumference of the circle, which is calculated as

$$\text{circumference} = 2 * \text{PI} * \text{radius}$$

Write a program that demonstrates the `Circle` class by asking the user for the circle’s radius, creating a `Circle` object, then reports the circle’s area, diameter, and circumference.

10. Pet Class

Design a class named `Pet`, which should have the following fields:

- `name`. The `name` field holds the name of a pet.
- `type`. The `type` field holds the type of animal that a pet is.
Example values are “Dog”, “Cat”, and “Bird”.
- `age`. The `age` field holds the pet’s age.

The `Pet` class should also have the following methods:

- `setName`. The `setName` method stores a value in the `name` field.
- `setType`. The `setType` method stores a value in the `type` field.

- `setAge`. The `setAge` method stores a value in the `age` field.
- `getName`. The `getName` method returns the value of the `name` field.
- `gettype`. The `getType` method returns the value of the `type` field.
- `getAge`. The `getAge` method returns the value of the `age` field.

Once you have designed the class, design a program that creates an object of the class and prompts the user to enter the name, type, and age of his or her pet. This data should be stored in the object. Use the object's accessor methods to retrieve the pet's name, type, and age and display this data on the screen.

11. Patient Charges

Write a class named `Patient` that has fields for the following data:

- First name, middle name, and last name
- Street address, city, state, and ZIP code
- Phone number
- Name and phone number of emergency contact

The `Patient` class should have a constructor that accepts an argument for each field. The `Patient` class should also have accessor and mutator methods for each field.

Next, write a class named `Procedure` that represents a medical procedure that has been performed on a patient. The `Procedure` class should have fields for the following data:

- Name of the procedure
- Date of the procedure
- Name of the practitioner who performed the procedure

- Charges for the procedure

The Procedure class should have a constructor that accepts an argument for each field. The Procedure class should also have accessor and mutator methods for each field.

Next, write a program that creates an instance of the Patient class, initialized with sample data. Then, create three instances of the Procedure class, initialized with the following data:

Procedure #1:

Procedure name: Physical Exam

Date: Today's date

Practitioner: Dr. Irvine

Charge: 250.00

Procedure #2:

Procedure name: X-ray

Date: Today's date

Practitioner: Dr. Jamison

Charge: 500.00

Procedure #3:

Procedure name: Blood test

Date: Today's date

Practitioner: Dr. Smith

Charge: 200.00

The program should display the patient's information, information about all three of the procedures, and the total charges of the three procedures.

Chapter 4 Decision Structures

Topics

1. [4.1 The `if` Statement](#)
2. [4.2 The `if-else` Statement](#)
3. [4.3 The `Payroll` Class](#)
4. [4.4 Nested `if` Statements](#)
5. [4.5 The `if-else-if` Statement](#)
6. [4.6 Logical Operators](#)
7. [4.7 Comparing `String` Objects](#)
8. [4.8 More about Variable Declaration and Scope](#)
9. [4.9 The Conditional Operator \(Optional\)](#)
10. [4.10 The `switch` Statement](#)
11. [4.11 Focus on Problem Solving: The `SalesCommission` Class](#)
12. [4.12 Generating Random Numbers with the `Random` Class](#)
13. [4.13 Common Errors to Avoid](#)

4.1 The if Statement

Concept:

The `if` statement is used to create a decision structure, which allows a program to have more than one path of execution. The `if` statement causes one or more statements to execute only when a boolean expression is true.



VideoNote The `if` Statement

In all the methods you have written so far, the statements are executed one after the other, in the order they appear. You might think of sequentially executed statements as the steps you take as you walk down a road. To complete the journey, you must start at the beginning and take each step, one after the other, until you reach your destination. This is illustrated in [Figure 4-1](#).

Figure 4-1 Sequence structure



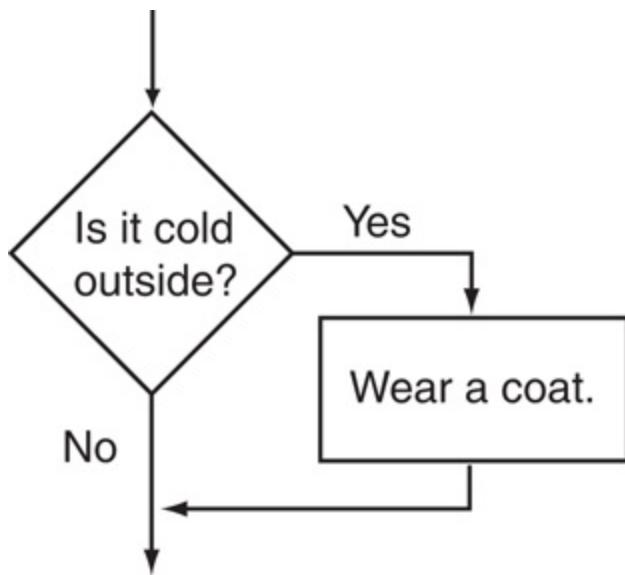
```
public class SquareArea
{
    public static void main(String[] args)
    {
        double length, width, area;
        Step 1 -----> length = 10;
        Step 2 -----> width = 5;
        Step 3 -----> area = length * width;
        Step 4 -----> System.out.print("The area is " + area);
    }
}
```

[Figure 4-1 Full Alternative Text](#)

The type of code in [Figure 4-1](#) is called a *sequence structure* because the statements are executed in sequence, without branching off in another direction. Programs often need more than one path of execution, however. Many algorithms require a program to execute some statements only under certain circumstances. This can be accomplished with a *decision structure*.

In a decision structure's simplest form, a specific action is taken only when a condition exists. If the condition does not exist, the action is not performed. The flowchart in [Figure 4-2](#) shows the logic of a decision structure. The diamond symbol represents a yes/no question or a true/false condition. If the answer to the question is “yes” (or if the condition is true), the program flow follows one path that leads to an action being performed. If the answer to the question is “no” (or the condition is false), the program flow follows another path that skips the action.

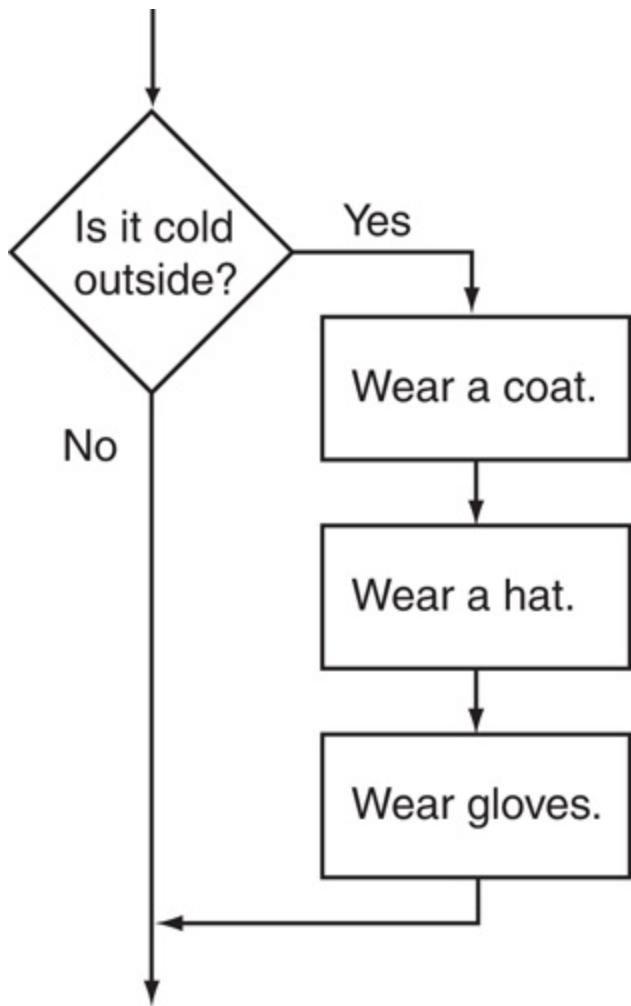
Figure 4-2 Simple decision structure logic



[Figure 4-2 Full Alternative Text](#)

In the flowchart, the action “Wear a coat” is performed only when it is cold outside. If it is not cold outside, the action is skipped. The action “Wear a coat” is *conditionally executed* because it is performed only when a certain condition (cold outside) exists. [Figure 4-3](#) shows a more elaborate flowchart, where three actions are taken only when it is cold outside.

Figure 4-3 Three-action decision structure logic



[Figure 4-3 Full Alternative Text](#)

One way to code a decision structure in Java is with the `if` statement. Here is the general format of the `if` statement:

```
if (BooleanExpression)
    statement;
```

The `if` statement is simple in the way it works: The *BooleanExpression* that appears inside the parentheses must be a boolean expression. A *boolean expression* is one that is either true or false. If the boolean expression is true, the very next *statement* is executed. Otherwise, it is skipped. The *statement* is *conditionally executed* because it executes only under the condition that the expression in the parentheses is true.

Using Relational Operators to Form Conditions

Typically, the condition that is tested by an `if` statement is formed with a relational operator. A *relational operator* determines whether a specific relationship exists between two values. For example, the greater than operator (`>`) determines whether one value is greater than another. The equal to operator (`==`) determines whether two values are equal. [Table 4-1](#) lists all of the Java relational operators.

Table 4-1 Relational operators

Relational Operators (in Order of Precedence)	Meaning
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code>==</code>	Equal to
<code>!=</code>	Not equal to

All of the relational operators are binary, which means they use two operands. Here is an example of an expression using the greater than operator:

```
length > width
```

This expression determines whether `length` is greater than `width`. If `length` is greater than `width`, the value of the expression is `true`. Otherwise, the value of the expression is `false`. Because the expression can be only `true` or `false`, it is a boolean expression. The following expression uses the less than operator to determine whether `length` is less than `width`:

```
length < width
```

[Table 4-2](#) shows examples of several boolean expressions that compare the variables *x* and *y*.

Table 4-2 boolean expressions using relational operators

Expression	Meaning
<i>x</i> > <i>y</i>	Is <i>x</i> greater than <i>y</i> ?
<i>x</i> < <i>y</i>	Is <i>x</i> less than <i>y</i> ?
<i>x</i> >= <i>y</i>	Is <i>x</i> greater than or equal to <i>y</i> ?
<i>x</i> <= <i>y</i>	Is <i>x</i> less than or equal to <i>y</i> ?
<i>x</i> == <i>y</i>	Is <i>x</i> equal to <i>y</i> ?
<i>x</i> != <i>y</i>	Is <i>x</i> not equal to <i>y</i> ?

Two of the operators, `>=` and `<=`, test for more than one relationship. The `>=` operator determines whether the operand on its left is greater than or equal to the operand on the right. Assuming that *a* is 4, *b* is 6, and *c* is 4, both of the expressions *b* `>=` *a* and *a* `>=` *c* are true and *a* `>=` 5 is false. When using this operator, the `>` symbol must precede the `=` symbol, and there is no space between them. The `<=` operator determines whether the operand on its left is less than or equal to the operand on its right. Once again, assuming that *a* is 4, *b* is 6, and *c* is 4, both *a* `<=` *c* and *b* `<=` 10 are true, but *b* `<=` *a* is false. When using this operator, the `<` symbol must precede the `=` symbol, and there is no space between them.

The `==` operator determines whether the operand on its left is equal to the operand on its right. If both operands have the same value, the expression is true. Assuming that *a* is 4, the expression *a* `==` 4 is true, and the expression *a* `==` 2 is false. Notice the equality operator is two `=` symbols together. Don't confuse this operator with the assignment operator, which is one `=` symbol.

The `!=` operator is the not equal operator. It determines whether the operand on its left is not equal to the operand on its right, which is the opposite of the `==` operator. As before, assuming `a` is 4, `b` is 6, and `c` is 4, both `a != b` and `b != c` are true because `a` is not equal to `b` and `b` is not equal to `c`. However, `a != c` is false, because `a` is equal to `c`.

Putting It All Together

Let's look at an example of the `if` statement:

```
if (sales > 50000)
    bonus = 500.0;
```

This statement uses the `>` operator to determine whether `sales` is greater than 50,000. If the expression `sales > 50000` is true, the variable `bonus` is assigned 500.0. If the expression is false, however, the assignment statement is skipped. The program in [Code Listing 4-1](#) shows another example. The user enters three test scores, and the program calculates their average. If the average is greater than 95, the program congratulates the user on obtaining a high score.

Code Listing 4-1 (AverageScore.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the if statement.
5  */
6
7 public class AverageScore
8 {
9     public static void main(String[] args)
10    {
11         double score1,      // Score #1
12             score2,      // Score #2
13             score3,      // Score #3
```

```
14         average;      // Average score
15
16     // Create a Scanner object to read input.
17     Scanner keyboard = new Scanner(System.in);
18
19     System.out.println("This program averages " +
20                     "3 test scores.");
21
22     // Get the first score.
23     System.out.print("Enter score #1: ");
24     score1 = keyboard.nextDouble();
25
26     // Get the second score.
27     System.out.print("Enter score #2: ");
28     score2 = keyboard.nextDouble();
29
30     // Get the third score.
31     System.out.print("Enter score #3: ");
32     score3 = keyboard.nextDouble();
33
34     // Calculate and display the average score.
35     average = (score1 + score2 + score3) / 3.0;
36     System.out.println("The average is " + average);
37
38     // If the average is higher than 95, congratulate
39     // the user.
40     if (average > 95)
41         System.out.println("That's a great score!");
42     }
43 }
```

Program Output with Example Input Shown in Bold

This program averages 3 test scores.

Enter score #1: **82** 

Enter score #2: **76** 

Enter score #3: **91** 

The average is 83.0

Program Output with Example Input Shown in Bold

This program averages 3 test scores.

Enter score #1: **97** 

Enter score #2: **94** 

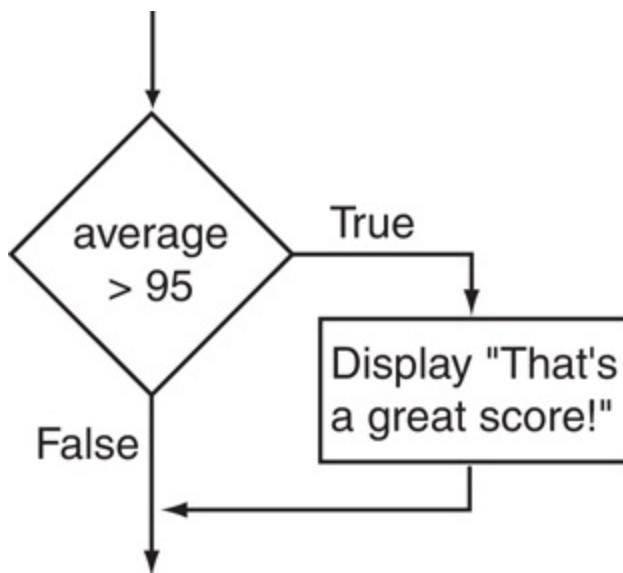
```
Enter score #3: 100 Enter  
The average is 97.0  
That's a great score!
```

The `if` statement in lines 40 and 41 causes the congratulatory message to be printed:

```
if (average > 95)  
    System.out.println("That's a great score!");
```

[Figure 4-4](#) shows the logic of this `if` statement.

Figure 4-4 Logic of the if statements



[Figure 4-4 Full Alternative Text](#)

[Table 4-3](#) shows other examples of `if` statements and their outcomes.

Table 4-3 Other examples of if

statements

Statement	Outcome
<pre>if (hours > 40) overtime = true;</pre>	If hours is greater than 40, assigns true to the boolean variable overtime.
<pre>if (value < 32) System.out.println("Invalid number");</pre>	If value is less than 32, displays the message "Invalid number"

Programming Style and the if Statement

Even though an `if` statement usually spans more than one line, it is really one long statement. For instance, the following `if` statements are identical except for the style in which they are written:

```
if (average > 95)
    System.out.println("That's a great score!");
if (average > 95) System.out.println("That's a great score!");
```

In both of these examples, the compiler considers the `if` statement and the conditionally executed statement as one unit, with a semicolon properly placed at the end. Indentations and spacing are for the human readers of a program, not the compiler. Here are two important style rules you should adopt for writing `if` statements:

- The conditionally executed statement should appear on the line after the `if` statement.
- The conditionally executed statement should be indented one level from

the `if` statement.

In most editors, each time you press the tab key, you are indenting one level. By indenting the conditionally executed statement, you are causing it to stand out visually. This is so you can tell at a glance what part of the program the `if` statement executes. This is a standard way of writing `if` statements, and is the method you should use.

Be Careful with Semicolons

You do not put a semicolon after the `if (BooleanExpression)` portion of an `if` statement, as illustrated in [Figure 4-5](#). This is because the `if` statement isn't complete without its conditionally executed statement.

Figure 4-5 Do not prematurely terminate an `if` statement with a semicolon

```
if (BooleanExpression)
    statement;
```

The diagram shows the code for an `if` statement. An arrow points from the word `statement` to the position where a semicolon would normally be placed. Another arrow points from the same position to the text `Semicolon goes here`. A third arrow points from the closing parenthesis of the `BooleanExpression` to the text `No semicolon here`.

[Figure 4-5 Full Alternative Text](#)

If you prematurely terminate an `if` statement with a semicolon, the compiler will not display an error message, but will assume that you are placing a *null statement* there. The *null statement*, which is an empty statement that does nothing, will become the conditionally executed statement. The statement intended to be conditionally executed will be disconnected from the `if` statement, and will always execute.

For example, look at the following code:

```
int x = 0, y = 10;

// The following if statement is prematurely
// terminated with a semicolon.
if (x > y);
    System.out.println(x + " is greater than " + y);
```

This code will always display the message “0 is greater than 10”. The `if` statement in this code is prematurely terminated with a semicolon. Because the `println` statement is not connected to the `if` statement, it will always execute.

Having Multiple Conditionally Executed Statements

The previous examples of the `if` statement conditionally execute a single statement. The `if` statement can also conditionally execute a group of statements, as long as they are enclosed in a set of braces. Enclosing a group of statements inside braces creates a *block* of statements. Here is an example:

```
if (sales > 50000)
{
    bonus = 500.0;
    commissionRate = 0.12;
    daysOff += 1;
}
```

If `sales` is greater than 50,000, this code will execute all three of the statements inside the braces, in the order they appear. If the braces were accidentally left out, however, the `if` statement conditionally executes only the very next statement. [Figure 4-6](#) illustrates this.

Figure 4-6 An if statement

missing its braces

```
if (sales > 50000)
    bonus = 500.0; ← Only this statement is
    commissionRate = 0.12; conditionally executed.
These statements are
    daysOff += 1; always executed.
```

[Figure 4-6 Full Alternative Text](#)

Flags

A *flag* is a boolean variable that signals when some condition exists in the program. When the flag variable is set to `false`, it indicates the condition does not yet exist. When the flag variable is set to `true`, it means the condition does exist.

For example, suppose a program similar to the previous test averaging program has a boolean variable named `highScore`. The variable might be used to signal that a high score has been achieved by the following code:

```
if (average > 95)
    highScore = true;
```

Later, the same program might use code similar to the following to test the `highScore` variable, in order to determine if a high score has been achieved:

```
if (highScore)
    System.out.println("That's a high score!");
```

You will find flag variables useful in many circumstances, and we will come back to them in future chapters.

Comparing Characters

You can use the relational operators to test character data as well as numbers. For example, the following code segment uses the `==` operator to compare the contents of the `char` variable `myLetter` to the character 'A':

```
char myLetter = 'A';
if (myLetter == 'A')
    System.out.println("That is the letter A.");
```

The `!=` operator can also be used with characters to test for inequality. For example, the following statement determines whether the `char` variable `myLetter` is not equal to the letter 'A':

```
if (myLetter != 'A')
    System.out.println("That is not the letter A.");
```

You can also use the `>`, `<`, `>=`, and `<=` operators to compare characters. Computers do not actually store characters in memory. Instead, they store numeric codes that represent the characters. Recall from [Chapter 2](#) that Java uses Unicode, which is a set of numbers that represents all the letters of the alphabet (both lowercase and uppercase), the printable digits 0 through 9, punctuation symbols, and special characters. When a character is stored in memory, it is actually the Unicode number that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.



Note:

Unicode is an international encoding system that is extensive enough to represent all the characters of all the world's alphabets.

In Unicode, letters are arranged in alphabetic order. Because 'A' comes before 'B', the numeric code for the character 'A' is less than the code for the character 'B'. (The code for 'A' is 65 and the code for 'B' is 66. [Appendix A](#) lists the codes for all of the printable English characters.) In the following `if` statement, the boolean expression '`'A' < 'B'`' is true.

```
if ('A' < 'B')
```

```
System.out.println("A is less than B.");
```

In Unicode, uppercase letters come before lowercase letters, so the numeric code for ‘A’ (65) is less than the numeric code for ‘a’ (97). In addition, the space character (code 32) comes before all the alphabetic characters.



Checkpoint

1. 4.1 Write an `if` statement that assigns 0 to `x` when `y` is equal to 20.
2. 4.2 Write an `if` statement that multiplies `payRate` by 1.5 if `hours` is greater than 40.
3. 4.3 Write an `if` statement that assigns 0.2 to `commission` if `sales` is greater than or equal to 10000.
4. 4.4 Write an `if` statement that sets the variable `fees` to 50 if the boolean variable `max` is `true`.
5. 4.5 Write an `if` statement that assigns 20 to the variable `y`, and assigns 40 to the variable `z`, if the variable `x` is greater than 100.
6. 4.6 Write an `if` statement that assigns 0 to the variable `b`, and assigns 1 to the variable `c`, if the variable `a` is less than 10.
7. 4.7 Write an `if` statement that displays “Goodbye” if the variable `myCharacter` contains the character ‘D’.

4.2 The if-else Statement

Concept:

The `if-else` statement will execute one group of statements if its boolean expression is true, or another group if its boolean expression is false.



VideoNote The `if-else` Statement

The `if-else` statement is an expansion of the `if` statement. Here is its general format:

```
if (BooleanExpression)
    statement or block
else
    statement or block
```

Like the `if` statement, a boolean expression is evaluated. If the expression is true, a statement or block of statements is executed. If the expression is false, however, a separate group of statements is executed. The program in [Code Listing 4-2](#) uses the `if-else` statement to handle a classic programming problem: division by zero. Division by zero is mathematically impossible to perform, and in Java it causes an error to occur at runtime.

Code Listing 4-2 (Division.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the if-else statement.
```

```

5  /*
6
7 public class Division
8 {
9     public static void main(String[] args)
10    {
11        int number1, number2; // Two numbers
12        double quotient; // The quotient of two numbers
13
14        // Create a Scanner object to read input.
15        Scanner keyboard = new Scanner(System.in);
16
17        // Get two numbers from the user.
18        System.out.print("Enter an integer: ");
19        number1 = keyboard.nextInt();
20        System.out.print("Enter another integer: ");
21        number2 = keyboard.nextInt();
22
23        // Determine whether division by zero will occur.
24        if (number2 == 0)
25        {
26            // Error - division by zero.
27            System.out.println("Division by zero is not possible");
28            System.out.println("Please run the program again and");
29            System.out.println("enter a number other than zero.");
30        }
31        else
32        {
33            // Perform the division and display the quotient.
34            quotient = (double) number1 / number2;
35            System.out.print("The quotient of " + number1);
36            System.out.print(" divided by " + number2);
37            System.out.println(" is " + quotient);
38        }
39    }
40 }

```

Program Output with Example Input Shown in Bold

Enter an integer: **10** 
 Enter another integer: **0** 
 Division by zero is not possible.
 Please run the program again and
 enter a number other than zero.

Program Output with Example Input Shown in Bold

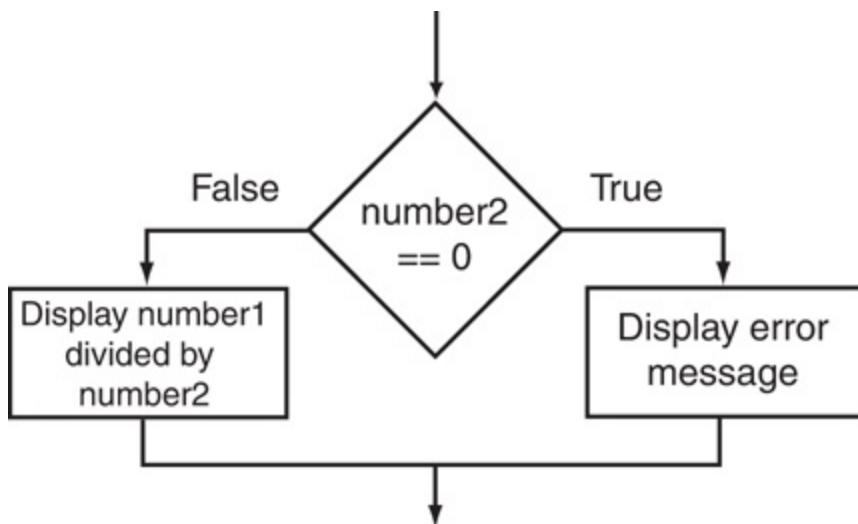
Enter an integer: 10 **Enter**

Enter another integer: 5 **Enter**

The quotient of 10 divided by 5 is 2.0

The value of `number2` is tested before the division is performed. If the user entered 0, the block of statements controlled by the `if` clause executes, displaying a message that indicates the program cannot perform a division by zero. Otherwise, the `else` clause takes control, which divides `number1` by `number2` and displays the result. [Figure 4-7](#) shows the logic of the `if-else` statement.

Figure 4-7 Logic of the if-else statement



[Figure 4-7 Full Alternative Text](#)



Checkpoint

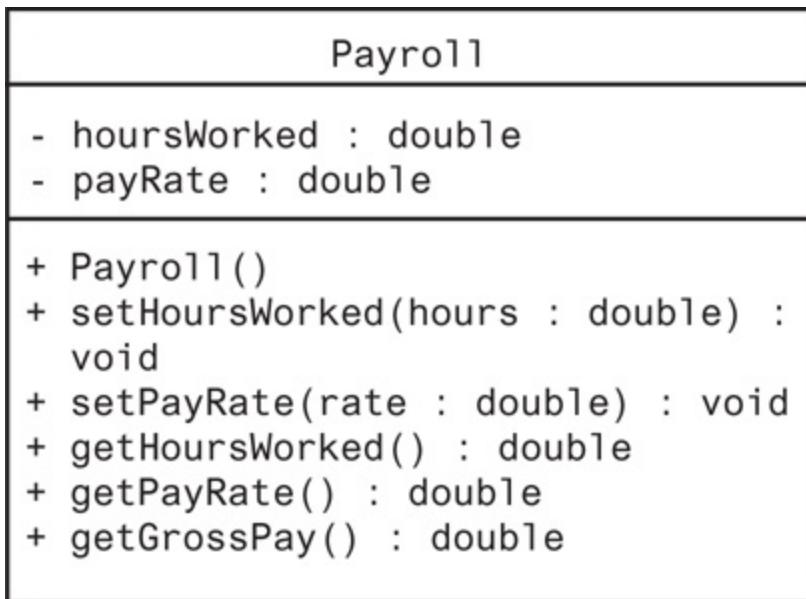
1. 4.8 Write an `if-else` statement that assigns 20 to the variable `y` if the variable `x` is greater than 100. Otherwise, it should assign 0 to the variable `y`.

2. 4.9 Write an `if-else` statement that assigns 1 to `x` when `y` is equal to 100. Otherwise, it should assign 0 to `x`.
3. Write an `if-else` statement that assigns 0.1 to `commission` unless `sales` is greater than or equal to 50000.0, in which case it assigns 0.2 to `commission`.
4. Write an `if-else` statement that assigns 0 to the variable `b`, and assigns 1 to the variable `c`, if the variable `a` is less than 10. Otherwise, it should assign -99 to the variable `b`, and assign 0 to the variable `c`.

4.3 The Payroll Class

In this section, we will examine a `Payroll` class that determines an employee's gross pay. The gross pay is calculated as the number of hours worked multiplied by the hourly pay rate. The class also has the ability to calculate overtime pay if more than 40 hours were worked. The employee earns 1.5 times his or her regular hourly pay rate for all hours over 40. [Figure 4-8](#) shows a UML diagram for the `Payroll` class.

Figure 4-8 UML diagram for the Payroll class



[Figure 4-8 Full Alternative Text](#)

Here is a summary of the class's fields.

- `hoursWorked` is a `double` that holds the number of hours the employee has worked.

- `payRate` is a `double` that holds the employee's hourly pay rate.

Here is a summary of the class's methods.

- A constructor initializes the fields to 0.0.
- `setHoursWorked` is a mutator method that accepts an argument and stores the argument's value in the `hoursworked` field.
- `setPayRate` is a mutator method that accepts an argument and stores the argument's value in the `payRate` field.
- `getHoursWorked` is an accessor method that returns the value in the `hoursworked` field.
- `getPayRate` is an accessor method that returns the value in the `payRate` field.
- The `getGrossPay` method calculates and returns the employee's gross pay. If the number of hours worked is greater than 40, the method adds overtime pay to the gross pay.

[Code Listing 4-3](#) shows the code for the `Payroll` class, which is stored in the file `Payroll.java`.

Code Listing 4-3 (Payroll.java)

```

1  /**
2   * This class holds values for hours worked and the
3   * hourly pay rate. It calculates the gross pay and
4   * adds additional pay for overtime.
5  */
6
7 public class Payroll
8 {
9     private double hoursWorked; // Number of hours worked
10    private double payRate;    // The hourly pay rate
11
12    /**
13     * The constructor initializes the hoursWorked and

```

```
14     * payRate fields to 0.0.  
15     */  
16  
17     public Payroll()  
18     {  
19         hoursWorked = 0.0;  
20         payRate = 0.0;  
21     }  
22  
23     /**  
24     * The setHoursWorked method accepts an argument  
25     * that is stored in the hoursWorked field.  
26     */  
27  
28     public void setHoursWorked(double hours)  
29     {  
30         hoursWorked = hours;  
31     }  
32  
33     /**  
34     * The setPayRate method accepts an argument that  
35     * is stored in the payRate field.  
36     */  
37  
38     public void setPayRate(double rate)  
39     {  
40         payRate = rate;  
41     }  
42  
43     /**  
44     * The getHoursWorked method returns the hoursWorked  
45     * field.  
46     */  
47  
48     public double getHoursWorked()  
49     {  
50         return hoursWorked  
51     }  
52  
53     /**  
54     * The getPayRate method returns the payRate field.  
55     */  
56  
57     public double getPayRate()  
58     {  
59         return payRate;  
60     }  
61
```

```

62     /**
63      * The getGrossPay method calculates and returns the
64      * gross pay. Overtime pay is also included.
65      */
66
67     public double getGrossPay()
68     {
69         double grossPay,           // Holds the gross pay
70                     overtimePay; // Holds pay for overtime
71
72         // Determine whether the employee worked more
73         // than 40 hours.
74         if (hoursWorked > 40)
75         {
76             // Calculate regular pay for the first 40 hours.
77             grossPay = 40 * payRate;
78
79             // Calculate overtime pay at 1.5 times the regular
80             // hourly pay rate.
81             overtimePay = (hoursWorked - 40) * (payRate * 1.5);
82
83             // Add the overtime pay to the regular pay.
84             grossPay += overtimePay;
85         }
86         else
87         {
88             // No overtime worked.
89             grossPay = payRate * hoursWorked;
90         }
91
92         return grossPay;
93     }
94 }
```

Notice that the `getGrossPay` method uses an `if-else` statement in lines 74 through 90 to control how the gross pay is calculated. If the `hoursWorked` field is greater than 40, the gross pay is calculated with the overtime pay included. Otherwise, the gross pay is calculated simply as `payRate` times `hoursWorked`. The program shown in [Code Listing 4-4](#) demonstrates the `Payroll` class.

Code Listing 4-4 (`GrossPay.java`)

```

1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program uses the Payroll class to
5  * calculate an employee's gross pay.
6 */
7
8 public class GrossPay
9 {
10     public static void main(String[] args)
11     {
12         double hours, // To hold hours worked
13             rate; // To hold the hourly pay rate
14
15         // Create a Scanner object to read input.
16         Scanner keyboard = new Scanner(System.in);
17
18         // Create a Payroll object.
19         Payroll employee = new Payroll();
20
21         // Get the number of hours worked.
22         System.out.print("How many hours did the " +
23                         "employee work? ");
24         hours = keyboard.nextDouble();
25
26         // Get the hourly pay rate.
27         System.out.print("What is the employee's " +
28                         "hourly pay rate? ");
29         rate = keyboard.nextDouble();
30
31         // Store the data.
32         employee.setHoursWorked(hours);
33         employee.setPayRate(rate);
34
35         // Display the gross pay.
36         System.out.println("The employee's gross pay " +
37                         "is $" + employee.getGrossPay());
38     }
39 }
```

Program Output with Example Input Shown in Bold

How many hours did the employee work? **30** 
 What is the employee's hourly pay rate? **20** 
 The employee's gross pay is \$600.0

Program Output with Example Input Shown in Bold

How many hours did the employee work? **50** 

What is the employee's hourly pay rate? **10** 
The employee's gross pay is \$550.0

4.4 Nested if Statements

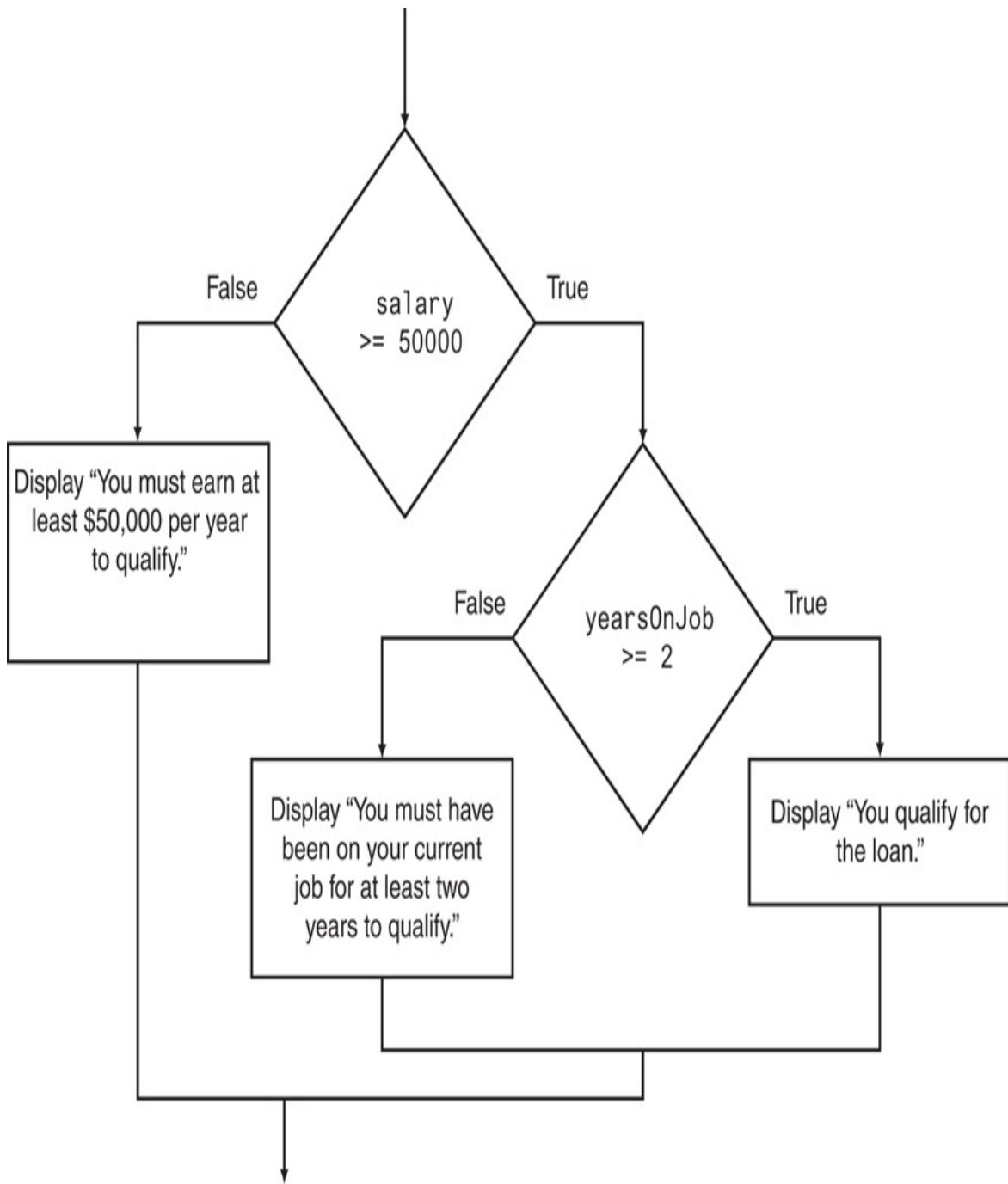
Concept:

To test more than one condition, an `if` statement can be nested inside another `if` statement.

Sometimes an `if` statement must be nested inside another `if` statement. For example, consider a banking program that determines whether a bank customer qualifies for a special low interest rate on a loan. To qualify, two conditions must exist: (1) the customer's salary must be at least \$50,000, and (2) the customer must have held his or her current job for at least 2 years.

[Figure 4-9](#) shows a flowchart for an algorithm that could be used in such a program.

Figure 4-9 Logic of nested if statements



[Figure 4-9 Full Alternative Text](#)

If we follow the flow of execution in the flowchart, we see that the expression `salary >= 50000` is tested. If this expression is false, there is no need to perform further tests; we know that the customer does not qualify for the special interest rate. If the expression is true, however, we need to test the second condition. This is done with a nested decision structure that tests the

expression `yearsOnJob >= 2`. If this expression is true, then the customer qualifies for the special interest rate. If this expression is false, then the customer does not qualify. [Code Listing 4-5](#) shows the complete program.

Code Listing 4-5 (LoanQualifier.java)

```
37                     "two years to qualify." );
38             }
39         }
40     else
41     {
42         System.out.println("You must earn at least " +
43                             "$50,000 per year to qualify.");
44     }
45 }
46 }
```

Program Output with Example Input Shown in Bold

Enter your annual salary: **55000.00** 

Enter the number of years at your current job: **1** 

You must have been on your current job for at least two years to

Program Output with Example Input Shown in Bold

Enter your annual salary: **25000.00** 

Enter the number of years at your current job: **5** 

You must earn at least \$50,000 per year to qualify.

Program Output with Example Input Shown in Bold

Enter your annual salary: **55000.00** 

Enter the number of years at your current job: **5** 

You qualify for the loan.

The first `if` statement (which begins in line 27) conditionally executes the second one (which begins in line 29). The only way the program will execute the second `if` statement is if the `salary` variable contains a value that is greater than or equal to 50,000. When this is the case, the second `if` statement tests the `yearsOnJob` variable. If it contains a value that is greater than or equal to 2, the program displays a message informing the user that he or she qualifies for the loan.

It should be noted that the braces used in the `if` statements in this program are not required. The statements could have been written as follows:

```
if (salary >= 50000)
    if (yearsOnJob >= 2)
        System.out.println("You qualify for the loan.");
    else
        System.out.println("You must have been on your " +
                           "current job for at least " +
                           "two years to qualify.");
else
    System.out.println("You must earn at least " +
                       "$50,000 per year to qualify.");
```

Not only do the braces make the statements easier to read, they also help in debugging code. When debugging a program with nested `if-else` statements, it is important to know to which `if` clause each `else` clause belongs. The rule for matching `else` clauses with `if` clauses is this: an `else` clause goes with the closest previous `if` clause that doesn't already have its own `else` clause. This is easy to see when the conditionally executed statements are enclosed in braces and are properly indented, as shown in [Figure 4-10](#). Each `else` clause lines up with the `if` clause it belongs to. These visual cues are important because nested `if` statements can be very long and complex.

Figure 4-10 Alignment of if and else clauses

```

if (salary >= 50000)
{
    if (yearsOnJob >= 2)
    {
        System.out.println("You qualify for the loan.");
    }
    else
    {
        System.out.println("You must have been on your " +
                           "current job for at least " +
                           "two years to qualify.");
    }
}
else
{
    System.out.println("You must earn at least " +
                       "$50,000 per year to qualify.");
}

```

This if and else go together.

This if and else go together.

[Figure 4-10 Full Alternative Text](#)

Testing a Series of Conditions

In the previous example, you saw how a program can use nested decision structures to test more than one condition. It is not uncommon for a program to have a series of conditions to test and then to perform an action, depending on which condition is true. One way to accomplish this is to have a decision structure with numerous other decision structures nested inside it. For example, consider the program presented in the following feature, *In the Spotlight*.

In the Spotlight: Multiple Nested Decision Structures



Suppose one of your professors uses the following 10-point grading scale for exams:

Test Score	Grade
90 and above	A
80–89	B
70–79	C
60–69	D
Below 60	F

Your professor has asked you to write a program that will allow the user to enter a test score then display the grade for that score. Here is the algorithm that you will use:

- Ask the user to enter a test score.

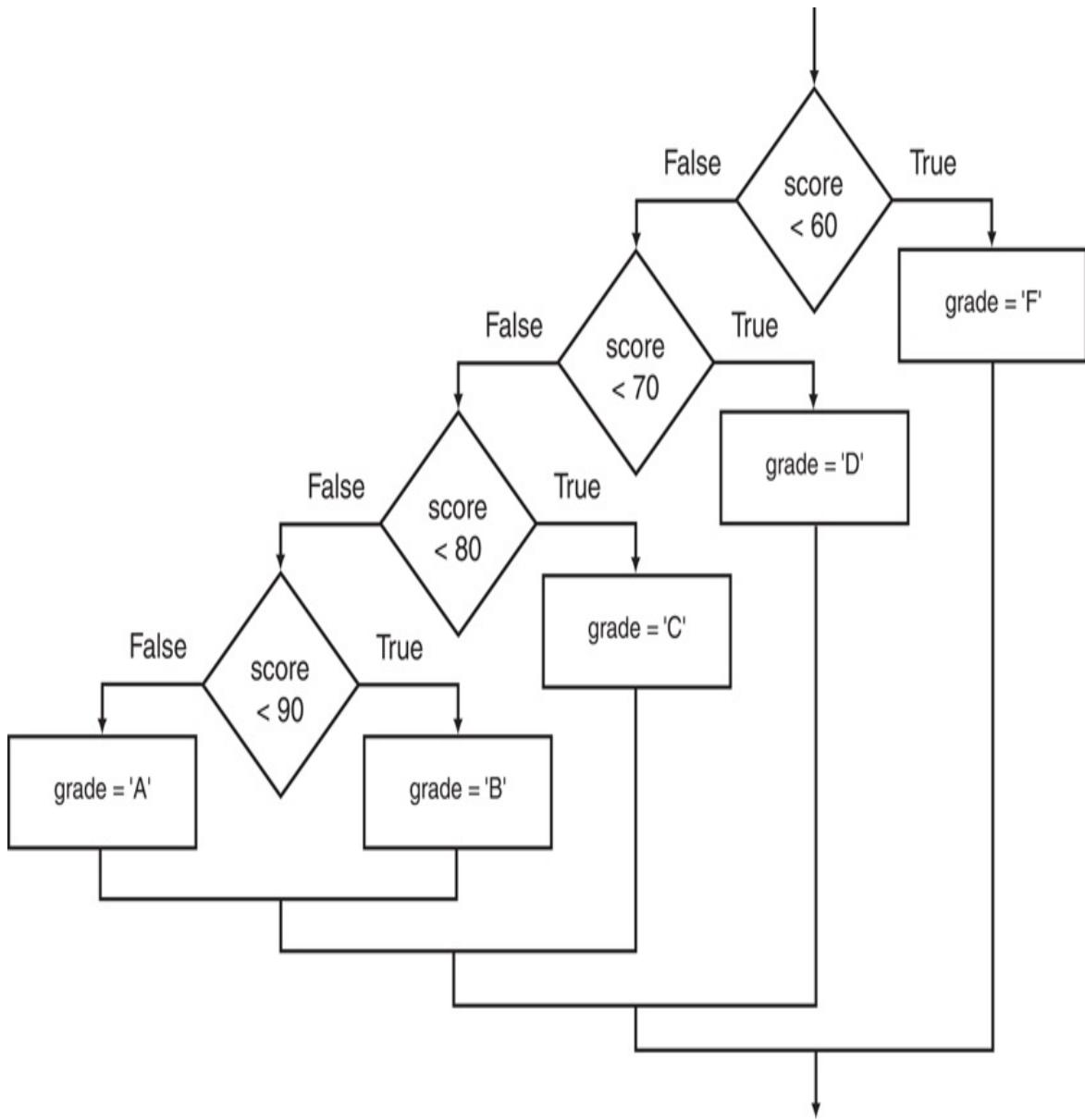
Determine the grade in the following manner:

If the score is less than 60, then the grade is F.

- Otherwise, if the score is less than 70, then the grade is D.
- Otherwise, if the score is less than 80, then the grade is C.
- Otherwise, if the score is less than 90, then the grade is B.
- Otherwise, the grade is A.

The process of determining the grade will require several nested decision structures, as shown in [Figure 4-11](#).

Figure 4-11 Nested decision structure to determine a grade

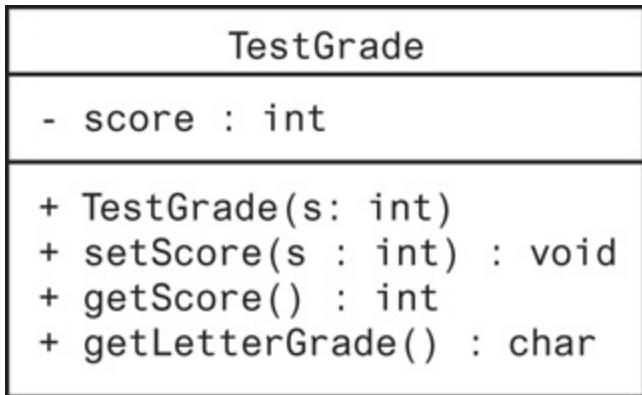


[Figure 4-11 Full Alternative Text](#)

You decide to create a class named `TestGrade` that will determine the letter grade for a given numeric test score. [Figure 4-12](#) shows the UML diagram for the class. Here is a summary of the class's fields and methods:

- `score`. A private int field to hold a numeric test score
- Constructor. Accepts an argument for the `score` field
- `setScore`. Mutator method that accepts an argument for the `score` field
- `getScore`. Accessor method that returns the value of the `score` field
- `getLetterGrade`. A method that uses the nested decision structures shown in [Figure 4-11](#) to determine a letter grade for the score, and returns that grade as a char.

Figure 4-12 UML diagram for the TestGrade class



[Figure 4-12 Full Alternative Text](#)

To use the `TestScore` class, you simply create an instance of it, passing a numeric test score as an argument to the constructor. You can call the `getLetterGrade` method to get the letter grade for that test score. [Code Listing 4-6](#) shows the code for the class. The nested decision structures appears in lines 48 through 76. [Code Listing 4-7](#) shows a complete program that demonstrates the class.

Code Listing 4-6 (TestGrade.java)

```
1  /**
2   * The TestGrade class determines a letter grade
3   * based on a numeric test score.
4   */
5
6  public class TestGrade
7  {
8      private int score;
9
10     /**
11      * The constructor accepts an argument
12      * for the score field.
13     */
14
15     public TestGrade(int s)
16     {
17         score = s;
18     }
19
20     /**
21      * The setScore method accepts an argument
22      * for the score field.
23     */
24
25     public void SetScore(int s)
26     {
27         score = s;
28     }
29
30     /**
31      * The getScore method returns the score field.
32     */
33
34     public int getScore()
35     {
36         return score;
37     }
38
39     /**
40      * The getLetterGrade returns the letter
41      * grade for the test score.
42     */
43
44     public char getLetterGrade()
45     {
46         char grade;
47
48         if (score < 60)
```

```

49         {
50             grade = 'F';
51         }
52     else
53     {
54         if (score < 70)
55         {
56             grade = 'D';
57         }
58     else
59     {
60         if (score < 80)
61         {
62             grade = 'C';
63         }
64     else
65     {
66         if (score < 90)
67         {
68             grade = 'B';
69         }
70     else
71     {
72         grade = 'A';
73     }
74     }
75 }
76 }
77
78     return grade;
79 }
80 }
```

Code Listing 4-7 (TestResults.java)

```

1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program uses the TestGrade class to determine
5  * a letter grade for a numeric test score.
6  */
7
8 public class TestResults
```

```

9  {
10     public static void main(String[] args)
11     {
12         int testScore;      // To hold a test score
13         char letterGrade; // To hold a letter grade
14
15         // Create a Scanner object to read input.
16         Scanner keyboard = new Scanner(System.in);
17
18         // Get the numeric test score.
19         System.out.print("Enter your numeric test score and " +
20                         "I will tell you the grade: ");
21         testScore = keyboard.nextInt();
22
23         // Create a TestGrade object with the numeric score.
24         TestGrade test = new TestGrade(testScore);
25
26         // Get the letter grade.
27         letterGrade = test.getLetterGrade();
28
29         // Display the grade.
30         System.out.print("Your grade is " +
31                         test.getLetterGrade());
32     }
33 }
```

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will tell you the grade: **80** 
Your grade is B

Program Output with Example Input Shown in Bold

Enter your numeric test score and I will tell you the grade: **72** 
Your grade is C



Checkpoint

- 4.12 Write nested **if** statements that perform the following test: If **amount1** is greater than 10 and **amount2** is less than 100, display the greater of the two.

2. 4.13 Write code that tests the variable x to determine whether it is greater than 0. If x is greater than 0, the code should test the variable y to determine whether it is less than 20. If y is less than 20, the code should assign 1 to the variable z . If y is not less than 20, the code should assign 0 to the variable z .

4.5 The if-else-if Statement

Concept:

The `if-else-if` statement tests a series of conditions. It is often simpler to test a series of conditions with the `if-else-if` statement than with a set of nested `if-else` statements.



VideoNote The if-else-if Statement

Even though the `TestGrade` class is a simple example (see [Code Listing 4-6](#)), the logic of the nested decision structure in lines 48 through 76 is fairly complex. You can alternatively test a series of conditions using the `if-else-if` statement. The `if-else-if` statement makes certain types of nested decision logic simpler to write. Here is the general format of the `if-else-if` statement:

```
if (expression_1)
{
    statement
    statement
    etc.
}
```

If *expression_1* is true these statements are executed, and the rest of the structure is ignored.

```
else if (expression_2)
{
    statement
    statement
    etc.
}
```

Otherwise, if *expression_2* is true these statements are executed, and the rest of the structure is ignored.

Insert as many else if clauses as necessary

```
else
{
    statement
    statement
    etc.
}
```

These statements are executed if none of the expressions above are true.

4.5-4 Full Alternative Text

When the statement executes, *expression_1* is tested. If *expression_1* is true, the block of statements that immediately follows is executed, and the rest of the structure is ignored. If *expression_1* is false, however, the program jumps to the very next else if clause and tests *expression_2*. If it is true, the block of statements that immediately follows is executed, then the rest of the structure is ignored. This process continues, from the top of the structure to the bottom, until one of the expressions is found to be true. If none of the expressions are true, the last else clause takes over, and the block of statements immediately following it is executed.

The last else clause, which does not have an if statement following it, is

referred to as the *trailing else*. The trailing else is optional, but in most cases, you will use it.



Note:

The general format shows braces surrounding each block of conditionally executed statements. As with other forms of the `if` statement, the braces are required only when more than one statement is conditionally executed.

The `TestGrade2` class shown in [Code Listing 4-8](#) demonstrates the `if-else-if` statement. This class is a modification of the `TestGrade` class shown in [Code Listing 4-6](#), which appears in the previous *In the Spotlight* feature.

Code Listing 4-8 (TestGrade2.java)

```
1  /**
2   * The TestGrade2 class determines a letter grade
3   * based on a numeric test score.
4   */
5
6  public class TestGrade2
7  {
8      private int score;
9
10     /**
11      * The constructor accepts an argument
12      * for the score field.
13     */
14
15     public TestGrade2(int s)
16     {
17         score = s;
18     }
19
20     /**
21      * The setScore method accepts an argument
22      * for the score field.
23     */
24 }
```

```

25     public void SetScore(int s)
26     {
27         score = s;
28     }
29
30     /**
31      * The getScore method returns the score field.
32      */
33
34     public int getScore()
35     {
36         return score;
37     }
38
39     /**
40      * The getLetterGrade returns the letter
41      * grade for the test score.
42      */
43
44     public char getLetterGrade()
45     {
46         char grade;
47
48         if (score < 60)
49             grade = 'F';
50         else if (score < 70)
51             grade = 'D';
52         else if (score < 80)
53             grade = 'C';
54         else if (score < 90)
55             grade = 'B';
56         else
57             grade = 'A';
58
59         return grade;
60     }
61 }
```

Let's analyze how the **if-else-if** statement in lines 48 through 57 works. First, the expression `score < 60` is tested in line 48:

```

→ if (score < 60)
    grade = 'F';
else if (score < 70)
    grade = 'D';
else if (score < 80)
```

```
    grade = 'C';
else if (score < 90)
    grade = 'B';
else
    grade = 'A';
```

If score is less than 60, the grade variable is assigned 'F' and the rest of the if-else-if statement is skipped. If score is not less than 60, the else clause in line 50 takes over and causes the next if statement to be executed:

```
if (score < 60)
    grade = 'F';
→ else if (score < 70)
    grade = 'D';
else if (score < 80)
    grade = 'C';
else if (score < 90)
    grade = 'B';
else
    grade = 'A';
```

The first if statement handled all the grades less than 60, so when this if statement executes, score will have a value of 60 or greater. If score is less than 70, the grade variable is assigned 'D' and the rest of the if-else-if statement is skipped. This chain of events continues until one of the expressions is found to be true, or the last else clause at the end of the statement is encountered. Notice the alignment and indentation that is used with the if-else-if statement: The starting if clause, the else if clauses, and the trailing else clause are all aligned, and the conditionally executed statements are indented.

The if-else-if Statement Compared to a Nested Decision Structure

You never have to use the if-else-if statement because its logic can be coded with nested if-else statements. However, a long series of nested if-

`else` statements has two particular disadvantages when you are debugging code:

- The code can grow complex and become difficult to understand.
- Because indenting is important in nested statements, a long series of nested `if-else` statements can become too long to be displayed on the computer screen without horizontal scrolling. Also, long statements tend to wrap around when printed on paper, making the code even more difficult to read.

The logic of an `if-else-if` statement is usually easier to follow than that of a long series of nested `if-else` statements. And, because all the clauses are aligned in an `if-else-if` statement, the lengths of the lines in the statement tend to be shorter.



Checkpoint

1. 4.14 What will the following program display?

```
public class CheckPoint
{
    public static void main(String[] args)
    {
        int funny = 7, serious = 15;
        funny = serious % 2;
        if (funny != 1)
        {
            funny = 0;
            serious = 0;
        }
        else if (funny == 2)
        {
            funny = 10;
            serious = 10;
        }
        else
        {
            funny = 1;
            serious = 1;
```

```

        }
    System.out.println(funny + " " + serious);
}
}

```

2. 4.15 The following program is used in a bookstore to determine how many discount coupons a customer gets. Complete the table that appears after the program.

```

import java.util.Scanner;
public class CheckPoint
{
    public static void main(String[] args)
    {
        int books, coupons;

        Scanner keyboard = new Scanner(System.in);
        System.out.print("How many books are being purchased? "
books = keyboard.nextInt();

        if (books < 1)
            coupons = 0;
        else if (books < 3)
            coupons = 1;
        else if (books < 5)
            coupons = 2;
        else
            coupons = 3;

        System.out.println("Number of coupons: " +
                           coupons);
    }
}

```

If the customer purchases this many books ...	This many coupons are given.
--	---

- 1
- 2
- 3
- 4
- 5
- 10

4.6 Logical Operators

Concept:

Logical operators connect two or more relational expressions into one or reverse the logic of an expression.

Java provides two binary logical operators, `&&` and `||`, which are used to combine two boolean expressions into a single expression. It also provides the unary `!` operator, which reverses the truth of a boolean expression. [Table 4-4](#) describes these logical operators.

Table 4-4 Logical operators

Operator Meaning		Effect
<code>&&</code>	AND	Connects two boolean expressions into one. Both expressions must be true for the overall expression to be true.
<code> </code>	OR	Connects two boolean expressions into one. One or both expressions must be true for the overall expression to be true. It is only necessary for one to be true, and it does not matter which one.
<code>!</code>	NOT	The <code>!</code> operator reverses the truth of a boolean expression. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

[Table 4-5](#) shows examples of several boolean expressions that use logical

operators.

Table 4-5 boolean expressions using logical operators

Expression	Meaning
<code>x > y && a < b</code>	Is x greater than y AND is a less than b?
<code>x == y x == z</code>	Is x equal to y OR is x equal to z?
<code>!(x > y)</code>	Is the expression <code>x > y</code> NOT true?

Let's take a close look at each of these operators.

The `&&` Operator

The `&&` operator is known as the logical AND operator. It takes two boolean expressions as operands and creates a boolean expression that is true only when both subexpressions are true. Here is an example of an `if` statement that uses the `&&` operator:

```
if (temperature < 20 && minutes > 12)
{
    System.out.println("The temperature is in the " +
                       "danger zone.");
}
```

In this statement, the two boolean expressions `temperature < 20` and `minutes > 12` are combined into a single expression. The message will be displayed only if `temperature` is less than 20 AND `minutes` is greater than 12. If either boolean expression is false, the entire expression is false and the message is not displayed.

[Table 4-6](#) shows a truth table for the `&&` operator. The truth table lists all the possible combinations of values that two expressions may have, and the resulting value returned by the `&&` operator connecting the two expressions.

Table 4-6 Truth table for the && operator

Expression	Resulting Value
true && false	false
false && true	false
false && false	false
true && true	true

As the table shows, both subexpressions must be `true` for the `&&` operator to return a `true` value.

The `&&` operator performs *short-circuit evaluation*. Here's how it works: If the expression on the left side of the `&&` operator is `false`, the expression on the right side will not be checked. Because the entire expression is `false` if only one of the subexpressions is `false`, it would waste CPU time to check the remaining expression. So, when the `&&` operator finds that the expression on its left is `false`, it short-circuits and does not evaluate the expression on its right.

The `&&` operator can be used to simplify programs that otherwise would use nested `if` statements. [Code Listing 4-9](#) shows another loan qualifying program. This one is written to use the `&&` operator.

Code Listing 4-9 (LogicalAnd.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the logical && operator.
5  */
6
7 public class LogicalAnd
8 {
9     public static void main(String[] args)
10    {
```

```

11     double salary;          // Annual salary
12     double yearsOnJob;     // Years at current job
13
14     // Create a Scanner object for keyboard input.
15     Scanner keyboard = new Scanner(System.in);
16
17     // Get the user's annual salary.
18     System.out.print("Enter your annual salary: ");
19     salary = keyboard.nextDouble();
20
21     // Get the number of years at the current job.
22     System.out.print("Enter the number of years " +
23                     "at your current job: ");
24     yearsOnJob = keyboard.nextDouble();
25
26     // Determine whether the user qualifies for the loan.
27     if (salary >= 50000 && yearsOnJob >= 2)
28     {
29         System.out.println("You qualify for the loan.");
30     }
31     else
32     {
33         System.out.println("You do not qualify.");
34     }
35 }
36 }
```

Program Output with Example Input Shown in Bold

Enter your annual salary: **55000.00** 
 Enter the number of years at your current job: **1** 
 You do not qualify.

Program Output with Example Input Shown in Bold

Enter your annual salary: **55000.00** 
 Enter the number of years at your current job: **4** 
 You qualify for the loan.

The message "You qualify for the loan." is displayed only when both the expressions `salary >= 50000` and `yearsOnJob >= 2` are true. If either of these expressions is false, the message "You do not qualify." is displayed.

You can also use logical operators with boolean variables. For example, assuming that `isValid` is a boolean variable, the following `if` statement determines whether `isValid` is true and `x` is greater than 90.

```
if (isValid && x > 90)
```

The || Operator

The `||` operator is known as the logical OR operator. It takes two boolean expressions as operands and creates a boolean expression that is true when either of the subexpressions are true. Here is an example of an `if` statement that uses the `||` operator:

```
if (temperature < 20 || temperature > 100)
{
    System.out.println("The temperature is in the " +
                       "danger zone.");
}
```

The message will be displayed if `temperature` is less than 20 OR `temperature` is greater than 100. If either relational test is true, the entire expression is true. [Table 4-7](#) shows a truth table for the `||` operator.

Table 4-7 Truth table for the || operator

Expression	Resulting Value
<code>true false</code>	<code>true</code>
<code>false true</code>	<code>true</code>
<code>false false</code>	<code>false</code>
<code>true true</code>	<code>true</code>

All it takes for an OR expression to be true is for one of the subexpressions to be true. It doesn't matter if the other subexpression is `false` or `true`. Like the `&&` operator, the `||` operator performs short-circuit evaluation. If the

subexpression on the left side of the `||` operator is true, the expression on the right side will not be checked. Because it is only necessary for one of the subexpressions to be true, it would waste CPU time to check the remaining expression.

The program in [Code Listing 4-10](#) is a different version of the previous program, shown in [Code Listing 4-9](#). This version uses the `||` operator to determine whether `salary >= 50000` is true *OR* `yearsOnJob >= 2` is true. If either expression is true, then the person qualifies for the loan.

Code Listing 4-10 (LogicalOr.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the logical || operator.
5  */
6
7 public class LogicalOr
8 {
9     public static void main(String[] args)
10    {
11        double salary;          // Annual salary
12        double yearsOnJob;    // Years at current job
13
14        // Create a Scanner object for keyboard input.
15        Scanner keyboard = new Scanner(System.in);
16
17        // Get the user's annual salary.
18        System.out.print("Enter your annual salary: ");
19        salary = keyboard.nextDouble();
20
21        // Get the number of years at the current job.
22        System.out.print("Enter the number of years " +
23                         "at your current job: ");
24        yearsOnJob = keyboard.nextDouble();
25
26        // Determine whether the user qualifies for the loan.
27        if (salary >= 50000 || yearsOnJob >= 2)
28        {
29            System.out.println("You qualify for the loan.");
30        }
31    else
```

```
32     {  
33         System.out.println("You do not qualify.");  
34     }  
35 }  
36 }
```

Program Output with Example Input Shown in Bold

Enter your annual salary: **20000.00**

Enter the number of years at your current job: **7**
You qualify for the loan.

Program Output with Example Input Shown in Bold

Enter your annual salary: **55000.00**

Enter the number of years at your current job: **1**
You qualify for the loan.

Program Output with Example Input Shown in Bold

Enter your annual salary: **20000.00**

Enter the number of years at your current job: **1**
You do not qualify.

The ! Operator

The ! operator performs a logical NOT operation. It is a unary operator that takes a boolean expression as its operand and reverses its logical value. In other words, if the expression is true, the ! operator returns false, and if the expression is false, it returns true. Here is an if statement using the ! operator:

```
if (!(temperature > 100))  
    System.out.println("This is below the maximum temperature.");
```

First, the expression (temperature > 100) is tested and a value of either true or false is the result. Then the ! operator is applied to that value. If the expression (temperature > 100) is true, the ! operator returns false. If the

expression (`temperature > 100`) is false, the `!` operator returns `true`. The previous code is equivalent to asking: “Is the temperature not greater than 100?”

[Table 4-8](#) shows a truth table for the `!` operator.

Table 4-8 Truth table for the `!` operator

Expression Resulting Value

<code>!true</code>	<code>false</code>
<code>!false</code>	<code>true</code>

The Precedence and Associativity of Logical Operators

Like other operators, the logical operators have orders of precedence and associativity. [Table 4-9](#) shows the precedence of the logical operators, from highest to lowest.

Table 4-9 Logical operators in order of precedence

!
&&
||

The `!` operator has a higher precedence than many of Java’s other operators. You should always enclose its operand in parentheses unless you intend to apply it to a variable or a simple expression with no other operators. For

example, consider the following expressions (assume `x` is an `int` variable with a value stored in it):

```
!(x > 2)  
!x > 2
```

The first expression applies the `!` operator to the expression `x > 2`. It is asking “is `x` not greater than 2?” The second expression, however, attempts to apply the `!` operator to `x` only. It is asking “is the logical complement of `x` greater than 2?” Because the `!` operator can be applied only to boolean expressions, this statement would cause a compiler error.

The `&&` and `||` operators rank lower in precedence than the relational operators, so precedence problems are less likely to occur. If you are unsure, however, it doesn’t hurt to use parentheses anyway.

```
(a > b) && (x < y) is the same as a > b && x < y  
(x == y) || (b > a) is the same as x == y || b > a
```

The logical operators evaluate their expressions from left to right. In the following expression, `a < b` is evaluated before `y == z`.

```
a < b || y == z
```

In the following expression, `y == z` is evaluated first, however, because the `&&` operator has higher precedence than `||`.

```
a < b || y == z && m > j
```

This expression is equivalent to:

```
(a < b) || ((y == z) && (m > j))
```

[Table 4-10](#) shows the precedence of all the operators we have discussed so far. This table includes the assignment, arithmetic, relational, and logical operators.

Table 4-10 Precedence of all

operators discussed so far

Order of Precedence	Operators	Description
1	- !	Unary negation, logical not
2	* / %	Multiplication, division, modulus
3	+ -	Addition, subtraction
4	< > <= >=	Less than, greater than, less than or equal to, greater than or equal to
5	== !=	Equal to, not equal to
6	&&	Logical AND
7		Logical OR
8	= += -= *= /= %=	Assignment and combined assignment

Checking Numeric Ranges with Logical Operators

Sometimes, you will need to write code that determines whether a numeric value is within a specific range of values or outside a specific range of values. When determining whether a number is inside a range, it's best to use the `&&` operator. For example, the following `if` statement checks the value in `x` to determine whether it is in the range of 20 through 40:

```
if (x >= 20 && x <= 40)
    System.out.println(x + " is in the acceptable range.");
```

The boolean expression in the `if` statement will be `true` only when `x` is greater than or equal to 20 AND less than or equal to 40. The value in `x` must be within the range of 20 through 40 for this expression to be `true`.

When determining whether a number is outside a range, the `||` operator is best to use. The following statement determines whether `x` is outside the

range of 20 through 40:

```
if (x < 20 || x > 40)
    System.out.println(x + " is outside the acceptable range.");
```

It's important not to get the logic of these logical operators confused. For example, the boolean expression in the following `if` statement would never test `true`:

```
if (x < 20 && x > 40)
    System.out.println(x + " is outside the acceptable range.");
```

Obviously, `x` cannot be less than 20 and at the same time be greater than 40.

Checkpoint

- 4.16 The following truth table shows various combinations of the values `true` and `false` connected by a logical operator. Complete the table by indicating if the result of such a combination is `true` or `false`.

Logical Expression Result (true or false)

```
true && false
true && true
false && true
false && false
true || false
true || true
false || true
false || false
!true
!false
```

- 4.17 Assume the variables `a = 2`, `b = 4`, and `c = 6`. Indicate by circling the T or F if each of the following conditions is true or false.

`a == 4 || b > 2` **T F**

`6 <= c && a > 3` **T F**

```
1 != b && c != 3 T F
a >= -1 || a <= b T F
!(a > 2)           T F
```

3. 4.18 Write an **if** statement that prints the message "The number is valid" if the variable speed is within the range 0 through 200.
4. 4.19 Write an **if** statement that prints the message "The number is not valid" if the variable speed is outside the range 0 through 200.

4.7 Comparing String Objects

Concept:

You should not use relational operators to compare `String` objects. Instead, you should use an appropriate `String` method.

You saw in the preceding sections how numeric values can be compared using the relational operators. You should not use the relational operators to compare `String` objects, however. Remember, a `String` variable holds the memory address of a `String` object. When you use a relational operator to compare two `String` variables, you are comparing the memory addresses that the variables contain, not the contents of the `String` objects that the variables reference.

To determine whether two `String` objects are equal, you should use the `String` class's `equals` method. The general form of the method is

```
StringReference1.equals(StringReference2)
```

`StringReference1` is a variable that references a `String` object, and `StringReference2` is another variable that references a `String` object. The method returns `true` if the two strings are equal, or `false` if they are not equal. Here is an example:

```
if (name1.equals(name2))
```

Assuming that `name1` and `name2` reference `String` objects, the expression in the `if` statement will return `true` if they are the same, or `false` if they are not the same. The program in [Code Listing 4-11](#) demonstrates the `equals` method.

Code Listing 4-11

(GoodStringCompare.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program correctly compares two String objects using
5  * the equals method.
6 */
7
8 public class GoodStringCompare
9 {
10     public static void main(String[] args)
11     {
12         String name1, name2, name3; // Three names
13
14         // Create a Scanner object to read input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Get a name.
18         System.out.print("Enter a name: ");
19         name1 = keyboard.nextLine();
20
21         // Get a second name.
22         System.out.print("Enter a second name: ");
23         name2 = keyboard.nextLine();
24
25         // Get a third name.
26         System.out.print("Enter a third name: ");
27         name3 = keyboard.nextLine();
28
29         // Compare name1 and name2
30         if (name1.equals(name2))
31         {
32             System.out.println(name1 + " and " + name2 +
33                                 " are the same.");
34         }
35         else
36         {
37             System.out.println(name1 + " and " + name2 +
38                                 " are NOT the same.");
39         }
40
41         // Compare name1 and name3
```

```
42     if (name1.equals(name3))
43     {
44         System.out.println(name1 + " and " + name3 +
45                             " are the same.");
46     }
47     else
48     {
49         System.out.println(name1 + " and " + name3 +
50                             " are NOT the same.");
51     }
52 }
53 }
```

Program Output with Example Input Shown in Bold

```
Enter a name: Mark 
Enter a second name: Mark 
Enter a third name: Mary 
Mark and Mark are the same.
Mark and Mary are NOT the same.
```

You can also compare `String` objects to string literals. Simply pass the string literal as the argument to the `equals` method, as shown here:

```
if (name1.equals("Mark"))
```

To determine if two strings are not equal, simply apply the `!` operator to the `equals` method's return value. Here is an example:

```
if (!name1.equals("Mark"))
```

The boolean expression in this `if` statement performs a not-equal-to operation. It determines whether the object referenced by `name1` is not equal to “Mark”.

The `String` class also provides the `compareTo` method, which can be used to determine whether one string is greater than, equal to, or less than another string. The general form of the method is:

`StringReference.compareTo(OtherString)`

`StringReference` is a variable that references a `String` object, and

otherString is either another variable that references a `String` object, or a string literal. The method returns an integer value that can be used in the following manner:

- If the method's return value is negative, the string referenced by *StringReference* (the calling object) is less than the *OtherString* argument.
- If the method's return value is 0, the two strings are equal.
- If the method's return value is positive, the string referenced by *StringReference* (the calling object) is greater than the *OtherString* argument.

For example, assume that `name1` and `name2` are variables that reference `String` objects. The following `if` statement uses the `compareTo` method to compare the strings.

```
if (name1.compareTo(name2) == 0)
    System.out.println("The names are the same.");
```

Also, the following expression compares the string referenced by `name1` to the string literal "Joe".

```
if (name1.compareTo("Joe") == 0)
    System.out.println("The names are the same.");
```

The program in [Code Listing 4-12](#) more fully demonstrates the `compareTo` method.

Code Listing 4-12 (`StringCompareTo.java`)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program compares two String objects using
5  * the compareTo method.
```

```

6  /*
7
8 public class StringComparison
9 {
10    public static void main(String[] args)
11    {
12        String name1, name2; // To hold two names
13
14        // Create a Scanner object to read input.
15        Scanner keyboard = new Scanner(System.in);
16
17        // Get a name.
18        System.out.print("Enter a name: ");
19        name1 = keyboard.nextLine();
20
21        // Get another name.
22        System.out.print("Enter another name: ");
23        name2 = keyboard.nextLine();
24
25        // Compare the names.
26        if (name1.compareTo(name2) < 0)
27        {
28            System.out.println(name1 + " is less than " + name2)
29        }
30        else if (name1.compareTo(name2) == 0)
31        {
32            System.out.println(name1 + " is equal to " + name2);
33        }
34        else if (name1.compareTo(name2) > 0)
35        {
36            System.out.println(name1 + " is greater than " + nam
37        }
38    }
39 }

```

Program Output with Example Input Shown in Bold

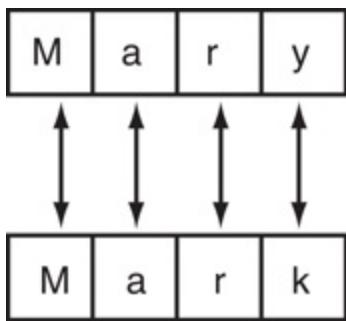
Enter a name: **Mary** 
 Enter another name: **Mark** 

Mary is greater than Mark

Let's take a closer look at this program. When you use the `compareTo` method to compare two strings, the strings are compared character by character. This is often called a *lexicographical comparison*. The program uses the `compareTo` method to compare the strings “Mary” and “Mark”, beginning

with the first or leftmost characters. This is illustrated in [Figure 4-13](#).

Figure 4-13 String comparison of “Mary” and “Mark”



Here is how the comparison takes place:

1. The “M” in “Mary” is compared with the “M” in “Mark.” Because these are the same, the next characters are compared.
2. The “a” in “Mary” is compared with the “a” in “Mark.” Because these are the same, the next characters are compared.
3. The “r” in “Mary” is compared with the “r” in “Mark.” Because these are the same, the next characters are compared.
4. The “y” in “Mary” is compared with the “k” in “Mark.” Because these are not the same, the two strings are not equal. The character “y” is greater than “k”, so it is determined that “Mary” is greater than “Mark.”

If one of the strings in a comparison is shorter in length than the other, Java can compare only the corresponding characters. If the corresponding characters are identical, then the shorter string is considered less than the longer string. For example, suppose the strings “High” and “Hi” were being compared. The string “Hi” would be considered less than “High” because it is shorter in length.

Ignoring Case in String Comparisons

The `equals` and `compareTo` methods perform case-sensitive comparisons, which means that uppercase letters are not considered the same as their lowercase counterparts. In other words, “A” is not the same as “a”. This can obviously lead to problems when you want to perform case-insensitive comparisons.

The `String` class provides the `equalsIgnoreCase` and `compareToIgnoreCase` methods. These methods work like the `equals` and `compareTo` methods, except the case of the characters in the strings is ignored. For example, the program in [Code Listing 4-13](#) asks the user to enter the “secret word,” which is similar to a password. The secret word is “PROSPERO”, and the program performs a case-insensitive string comparison to determine whether the user has entered it.

Code Listing 4-13 (SecretWord.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates a case-insensitive string compar
5  */
6
7 public class SecretWord
8 {
9     public static void main(String[] args)
10    {
11        String input; // To hold the user's input
12
13        // Create a Scanner object to read input.
14        Scanner keyboard = new Scanner(System.in);
15
16        // Prompt the user to enter the secret word.
```

```
17     System.out.print("Enter the secret word: ");
18     input = keyboard.nextLine();
19
20     // Determine if the user entered the secret word.
21     if (input.equalsIgnoreCase("PROSPERO"))
22     {
23         System.out.println("Congratulations! You know the "
24             "secret word!");
25     }
26     else
27     {
28         System.out.println("Sorry, that is NOT the " +
29             "secret word!");
30     }
31 }
32 }
```

Program Output with Example Input Shown in Bold

Enter the secret word: **Ferdinand** 
Sorry, that is NOT the secret word!

Program Output with Example Input Shown in Bold

Enter the secret word: **Prospero** 
Congratulations! You know the secret word!

See the file *CompareWithoutCase.java* for an example demonstrating the `compareToIgnoreCase` method. (The book's source code is available at www.pearsonhighered.com/gaddis.)



Checkpoint

1. 4.20 Assume the variable `name` references a `String` object. Write an `if` statement that displays “Do I know you?” if the `String` object contains “Timothy”.
2. 4.21 Assume the variables `name1` and `name2` reference two different `String` objects, containing different strings. Write code that displays the strings referenced by these variables in alphabetical order.

3. 4.22 Modify the statement you wrote in response to [Checkpoint 4.20](#) so it performs a case-insensitive comparison.

4.8 More about Variable Declaration and Scope

Concept:

The scope of a variable is limited to the block in which it is declared.

Recall from [Chapter 2](#) that a local variable is a variable that is declared inside a method. Java allows you to create local variables just about anywhere in a method. For example, look at the program in [Code Listing 4-14](#). The `main` method declares two `String` reference variables: `firstName` and `lastName`. Notice that the declarations of these variables appear near the code that first uses the variables.

Code Listing 4-14 (VariableScope.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates how variables may be declared
5  * in various locations throughout a program.
6 */
7
8 public class VariableScope
9 {
10     public static void main(String[] args)
11     {
12         // Create a Scanner object for keyboard input.
13         Scanner keyboard = new Scanner(System.in);
14
15         // Get the user's first name.
```

```
16     System.out.print("Enter your first name: ");
17     String firstName;
18     firstName = keyboard.nextLine();
19
20     // Get the user's last name.
21     System.out.print("Enter your last name: ");
22     String lastName;
23     lastName = keyboard.nextLine();
24
25     // Display a message.
26     System.out.println("Hello " + firstName +
27                         " " + lastName);
28 }
29 }
```

Although it is a common practice to declare all of a method's local variables at the beginning of the method, it is possible to declare them at later points. Sometimes programmers declare certain variables near the part of the program where they are used in order to make their purpose more evident.



Note:

When a program is running and it enters the section of code that constitutes a variable's scope, it is said that the variable “comes into scope.” This simply means that the variable is now visible and the program can reference it. Likewise, when a variable “leaves scope,” it cannot be used.

4.9 The Conditional Operator (Optional)

Concept:

You can use the conditional operator to create short expressions that work like `if-else` statements.

The *conditional operator* is powerful and unique. Because it takes three operands, it is considered a ternary operator. The conditional operator provides a shorthand method of expressing a simple `if-else` statement. The operator consists of the question mark (?) and the colon (:). You use the operator to write a conditional expression in the following format:

BooleanExpression ? *Value1* : *Value2*;

The *BooleanExpression* is like the boolean expression in the parentheses of an `if` statement. If the *BooleanExpression* is true, then the value of the conditional expression is *Value1*. Otherwise, the value of the conditional expression is *Value2*. Here is an example of a statement using the conditional operator:

```
y = x < 0 ? 10 : 20;
```

This preceding statement performs the same operation as the following `if-else` statement:

```
if (x < 0)
    y = 10;
else
    y = 20;
```

The conditional operator gives you the ability to pack decision-making power

into a concise line of code. With a little imagination, it can be applied to many other programming problems. For instance, consider the following statement:

```
System.out.println("Your grade is: " + (score < 60 ? "Fail." : "P
```

Converted to an if-else statement, it would be written as follows:

```
if (score < 60)
    System.out.println("Your grade is: Fail.");
else
    System.out.println("Your grade is: Pass.");
```



Note:

The parentheses are placed around the conditional expression because the + operator has higher precedence than the ?: operator. Without the parentheses, the + operator would concatenate the value in score with the string "Your grade is: ".



Checkpoint

1. 4.23 Rewrite the following if-else statements as conditional expressions.

1.

```
if (x > y)
    z = 1;
else
    z = 20;
```
2.

```
if (temp > 45)
    population = base * 10;
else
    population = base * 2;
```
3.

```
if (hours > 40)
    wages *= 1.5;
```

```
    else
        wages *= 1;

4. if (result >= 0)
    System.out.println("The result is positive.");
else
    System.out.println("The result is negative.")
```

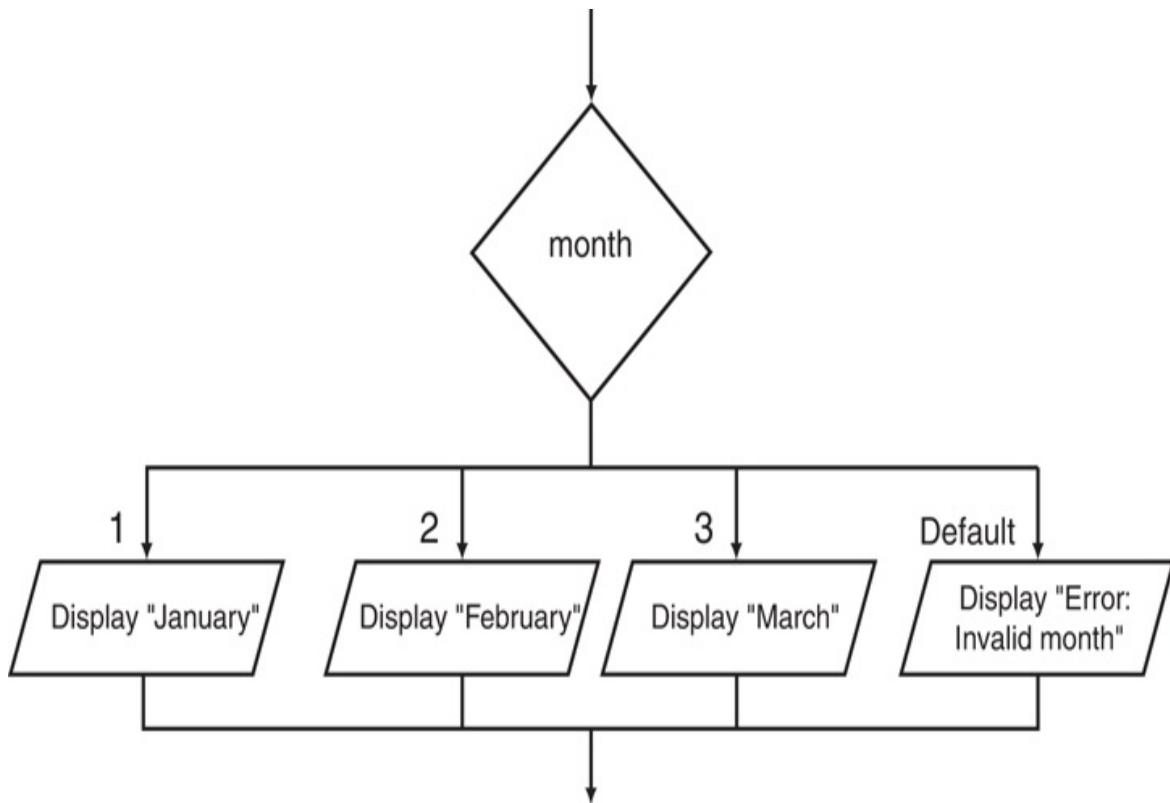
4.10 The switch Statement

Concept:

The `switch` statement lets the value of a variable or expression determine to where the program will branch.

The *switch statement* is a *multiple alternative decision structure*. It allows you to test the value of a variable or an expression, then use that value to determine which statement or set of statements to execute. [Figure 4-14](#) shows an example of how a multiple alternative decision structure looks in a flowchart.

Figure 4-14 A multiple alternative decision structure



[Figure 4-14 Full Alternative Text](#)

In the flowchart, the diamond symbol shows `month`, which is the name of a variable. If the `month` variable contains the value 1, the program displays *January*. If the `month` variable contains the value 2, the program displays *February*. If the `month` variable contains the value 3, the program displays *March*. If the `month` variable contains none of these values, the action that is labeled `Default` is executed. In this case, the program displays *Error: Invalid month*.

Here is the general format of a `switch` statement in Java:

```

switch (testExpression)
{
    case value_1:
        statement;
        statement;
        etc.
        break;           | These statements are executed
                        | if the testExpression is
                        | equal to value_1.

    case value_2:
        statement;
        statement;
        etc.
        break;           | These statements are executed
                        | if the testExpression is
                        | equal to value_2.

    Insert as many case sections as necessary.

    case value_N:
        statement;
        statement;
        etc.
        break;           | These statements are executed
                        | if the testExpression is
                        | equal to value_N.

    default:
        statement;
        statement;
        etc.
        break;           | These statements are executed
                        | if the testExpression is not
                        | equal to any of the case values.

}

```

The *testExpression* is a variable or expression.

These statements are executed if the *testExpression* is equal to *value_1*.

These statements are executed if the *testExpression* is equal to *value_2*.

These statements are executed if the *testExpression* is equal to *value_N*.

These statements are executed if the *testExpression* is not equal to any of the case values.

4.10-12 Full Alternative Text

The first line of the statement starts with the word `switch`, followed by a *testExpression*, which is enclosed in parentheses. The *testExpression* is a variable or an expression that gives a `char`, `byte`, `short`, `int`, or `String` value. (If you are using a version of Java prior to Java 7, the *testExpression*

cannot be a string.)

Beginning at the next line is a block of code enclosed in curly braces. Inside this block of code are one or more case sections. A case section begins with the word `case`, followed by a value, followed by a colon. Each case section contains one or more statements, followed by a `break` statement. After all the case sections, an optional default section appears.

When the `switch` statement executes, it compares the value of the `testExpression` with the values that follow each of the case statements (from top to bottom). When it finds a case value that matches the `testExpression`'s value, the program branches to the case statement. The statements that follow the case statement are executed, until a `break` statement is encountered. At that point, the program jumps out of the `switch` statement. If the `testExpression` does not match any of the case values, the program branches to the default statement and executes the statements that immediately follow it.



Note:

Each of the case values must be unique.

For example, the following code performs the same operation as the flowchart in [Figure 4-14](#). Assume `month` is an `int` variable.

```
switch (month)
{
    case 1:
        System.out.println("January");
        break;
    case 2:
        System.out.println("February");
        break;
    case 3:
        System.out.println("March");
        break;
    default:
        System.out.println("Error: Invalid month");
        break;
```

```
}
```

In this example, the *testExpression* is the `month` variable. The `month` variable is evaluated and one of the following actions takes place:

- If the value in the `month` variable is 1, the program branches to the `case 1:` section and executes the `System.out.println("January")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is 2, the program branches to the `case 2:` section and executes the `System.out.println("February")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is 3, the program branches to the `case 3:` section and executes the `System.out.println("March")` statement that immediately follows it. The `break` statement then causes the program to exit the `switch` statement.
- If the value in the `month` variable is not 1, 2, or 3, the program branches to the `default:` section and executes the `System.out.println("Error: Invalid month")` statement that immediately follows it.

The `switch` statement can be used as an alternative to an `if-else-if` statement that tests the same variable or expression to several different values. For example, the previously shown `switch` statement works like this `if-else-if` statement:

```
if (month == 1)
{
    System.out.println("January");
}
else if (month == 2)
{
    System.out.println("February");
}
else if (month == 3)
{
    System.out.println("March");
}
```

```
else
{
    System.out.println("Error: Invalid month");
}
```



Note:

The default section is optional. If you leave it out, however, the program has nowhere to branch to if the *testExpression* does not match any of the case values.

The program in [Code Listing 4-15](#) shows how a simple switch statement works.

Code Listing 4-15 (SwitchDemo.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates the switch statement.
5  */
6
7 public class SwitchDemo
8 {
9     public static void main(String[] args)
10    {
11         String input; // To hold keyboard input
12         char choice; // To store the user's choice
13
14         // Create a Scanner object to read input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Ask the user to enter A, B, or C.
18         System.out.print("Enter A, B, or C: ");
19         input = keyboard.nextLine();
20         choice = input.charAt(0); // Get the first char
21
22         // Determine which character the user entered.
```

```
23     switch (choice)
24     {
25         case 'A':
26             System.out.println("You entered A.");
27             break;
28         case 'B':
29             System.out.println("You entered B.");
30             break;
31         case 'C':
32             System.out.println("You entered C.");
33             break;
34     default:
35         System.out.println("That's not A, B, or C!");
36     }
37 }
38 }
```

Program Output with Example Input Shown in Bold

Enter A, B, or C: **B** 
You entered B.

Program Output with Example Input Shown in Bold

Enter A, B, or C: **F** 
That's not A, B, or C!

Notice the break statements that appear in lines 27, 30, and 33. The case statements show the program where to start executing in the block and the break statements show the program where to stop. Without the break statements, the program would execute all of the lines from the matching case statement to the end of the block.



Note:

The default section (or the last case section if there is no default) does not need a break statement. Some programmers prefer to put one there anyway for consistency.

The program in [Code Listing 4-16](#) is a modification of [Code Listing 4-15](#), without the break statements.

Code Listing 4-16 (NoBreaks.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates a switch statement
5  * without any break statements.
6  */
7
8 public class NoBreaks
9 {
10     public static void main(String[] args)
11     {
12         String input; // To hold keyboard input
13         char choice; // To store the user's choice
14
15         // Create a Scanner object to read input.
16         Scanner keyboard = new Scanner(System.in);
17
18         // Ask the user to enter A, B, or C.
19         System.out.print("Enter A, B, or C: ");
20         input = keyboard.nextLine();
21         choice = input.charAt(0); // Get the first char
22
23         // Determine which character the user entered.
24         switch (choice)
25         {
26             case 'A':
27                 System.out.println("You entered A.");
28             case 'B':
29                 System.out.println("You entered B.");
30             case 'C':
31                 System.out.println("You entered C.");
32             default:
33                 System.out.println("That's not A, B, or C!");
34         }
35     }
36 }
```

Program Output with Example Input Shown in Bold

Enter A, B, or C: **A** 
You entered A.
You entered B.
You entered C.
That's not A, B, or C!

Program Output with Example Input Shown in Bold

Enter A, C, or C: **c** 
You entered C.
That's not A, B, or C!

Without the break statement, the program “falls through” all of the statements below the one with the matching case expression. Sometimes this is what you want. For instance, the program in [Code Listing 4-17](#) asks the user to select a grade of pet food. The available choices are A, B, and C. The switch statement will recognize either uppercase or lowercase letters.

Code Listing 4-17 (PetFood.java)

```
1 import java.util.Scanner; // Needed for the Scanner class
2
3 /**
4  * This program demonstrates a switch statement.
5 */
6
7 public class PetFood
8 {
9     public static void main(String[] args)
10    {
11         String input; // To hold keyboard input
12         char feedGrade; // To hold the feed grade
13
14         // Create a Scanner object to read input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Get the desired pet food grade.
18         System.out.println("Our pet food is available in " +
19                             "three grades:");
20         System.out.print("A, B, and C. Which do you want " +
21                         "pricing for? ");
22         input = keyboard.nextLine();
```

```
23     feedGrade = input.charAt(0); // Get the first char.  
24  
25     // Determine the grade that was entered.  
26     switch(feedGrade)  
27     {  
28         case 'a':  
29         case 'A':  
30             System.out.println("30 cents per lb.");  
31             break;  
32         case 'b':  
33         case 'B':  
34             System.out.println("20 cents per lb.");  
35             break;  
36         case 'c':  
37         case 'C':  
38             System.out.println("15 cents per lb.");  
39             break;  
40     default:  
41         System.out.println("Invalid choice.");  
42     }  
43 }  
44 }
```

Program Output with Example Input Shown in Bold

Our dog food is available in three grades:

A, B, and C. Which do you want pricing for? **b** 
20 cents per lb.

Program Output with Example Input Shown in Bold

Our dog food is available in three grades:

A, B, and C. Which do you want pricing for? **B** 
20 cents per lb.

When the user enters ‘a’ the corresponding case has no statements associated with it, so the program falls through to the next case, which corresponds with ‘A’.

```
case 'a':  
case 'A':  
    System.out.println("30 cents per lb.");  
    break;
```

The same technique is used for ‘b’ and ‘c’.

If you are using a version of Java prior to Java 7, a switch statement’s *testExpression* can be a char, byte, short, or int value. Beginning with Java 7, however, the *testExpression* can also be a string. The program in [Code Listing 4-18](#) demonstrates this capability.

Code Listing 4-18 (Seasons.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program translates the English names of
5  * the seasons into Spanish.
6 */
7
8 public class Seasons
9 {
10     public static void main(String[] args)
11     {
12         String input;
13
14         // Create a Scanner object for keyboard input.
15         Scanner keyboard = new Scanner(System.in);
16
17         // Get a day from the user.
18         System.out.print("Enter the name of a season: ");
19         input = keyboard.nextLine();
20
21         // Translate the season to Spanish.
22         switch (input)
23         {
24             case "spring":
25                 System.out.println("la primavera");
26                 break;
27             case "summer":
28                 System.out.println("el verano");
29                 break;
30             case "autumn":
31             case "fall":
32                 System.out.println("el otono");
33                 break;
34             case "winter":
```

```
35         System.out.println("el invierno");
36         break;
37     default:
38         System.out.println("Please enter one of these words:
39                         spring, summer, autumn, fall,
40     }
41 }
42 }
```

Program Output with Example Input Shown in Bold

Enter the name of a season: **summer** 
el verano

Program Output with Example Input Shown in Bold

Enter the name of a season: **fall** 
el otono



Checkpoint

- 4.24 Complete the following program skeleton by writing a switch statement that displays “one” if the user has entered 1, “two” if the user has entered 2, and “three” if the user has entered 3. If a number other than 1, 2, or 3 is entered, the program should display an error message.

```
import java.util.Scanner;
public class CheckPoint
{
    public static void main(String[] args)
    {
        int userNum;
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter one of the numbers " +
                        "1, 2, or 3: ");
        userNum = keyboard.nextInt();
        //
        // Write the switch statement here.
        //
    }
}
```

2. 4.25 Rewrite the following if-else-if statement as a switch statement:

```
if (selection == 'A')
    System.out.println("You selected A.");
else if (selection == 'B')
    System.out.println("You selected B.");
else if (selection == 'C')
    System.out.println("You selected C.");
else if (selection == 'D')
    System.out.println("You selected D.");
else
    System.out.println("Not good with letters, eh?");
```

3. 4.26 Explain why you cannot convert the following if-else-if statement into a switch statement:

```
if (temp == 100)
    x = 0;
else if (population > 1000)
    x = 1;
else if (rate < .1)
    x = -1;
```

4. 4.27 What is wrong with the following switch statement?

```
// This code has errors!!!
switch (temp)
{
    case temp < 0 :
        System.out.println("Temp is negative.");
        break;
    case temp == 0:
        System.out.println("Temp is zero.");
        break;
    case temp > 0 :
        System.out.println("Temp is positive.");
        break;
}
```

5. 4.28 What will the following code display?

```
int funny = 7, serious = 15;
funny = serious * 2;
switch (funny)
{
```

```
case 0 :  
    System.out.println("That is funny.");  
    break;  
case 30:  
    System.out.println("That is serious.");  
    break;  
case 32:  
    System.out.println("That is seriously funny.");  
    break;  
default:  
    System.out.println(funny);  
}
```

4.11 Focus on Problem Solving: The SalesCommission Class

In this section, we will examine a case study that implements many of the topics discussed in this chapter. In addition, we will discuss how a lengthy algorithm can be decomposed into several shorter methods.

Hal's Home Computer Emporium is a retail seller of personal computers. Hal's sales staff work strictly on commission. At the end of the month, each salesperson's commission is calculated according to [Table 4-11](#).

Table 4-11 Sales commission rates

Sales This Month Commission Rate

Less than \$10,000	5%
\$10,000–14,999	10%
\$15,000–17,999	12%
\$18,000–21,999	14%
\$22,000 or more	16%

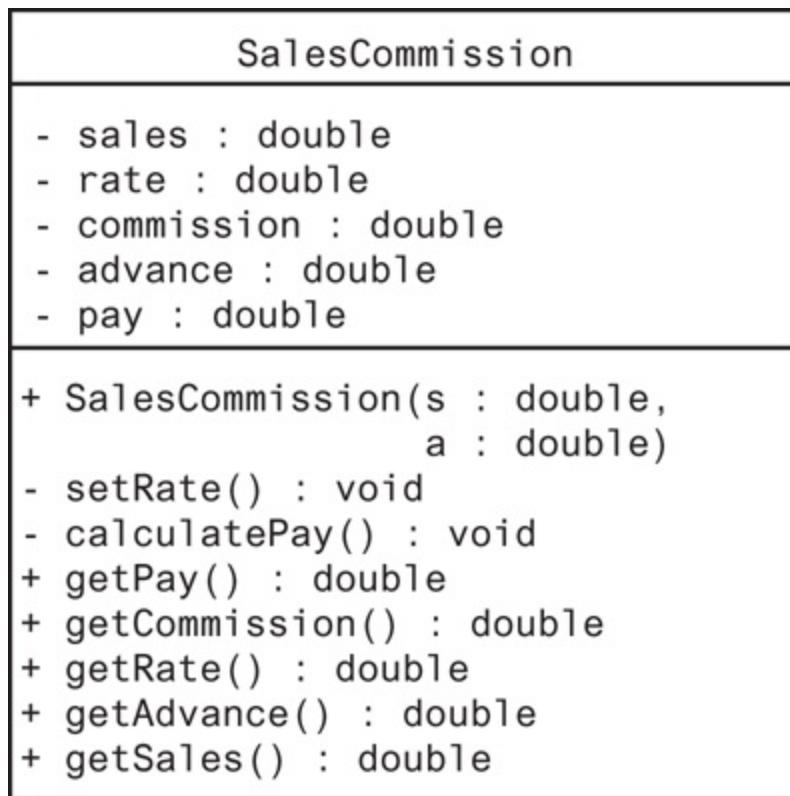
For example, a salesperson with \$16,000 in monthly sales will earn a 12 percent commission (\$1,920.00). Another salesperson with \$20,000 in monthly sales will earn a 14 percent commission (\$2,800.00).

Because the staff gets paid once per month, Hal allows each employee to take up to \$1,500 per month in advance. When sales commissions are calculated, the amount of each employee's advanced pay is subtracted from the commission. If any salesperson's commissions are less than the amount of their advance, they must reimburse Hal for the difference.

Here are two examples: Beverly and John have \$21,400 and \$12,600 in sales, respectively. Beverly's commission is \$2,996, and John's commission is \$1,260. Both Beverly and John took \$1,500 in advanced pay. At the end of the month, Beverly gets a check for \$1,496, but John must pay \$240 back to Hal.

Now we will examine a program that eases the task of calculating the end-of-month commission. The core of the program will be a `SalesCommission` class that holds the primary data for a salesperson, determines the rate of commission, and calculates the salesperson's pay. [Figure 4-15](#) shows a UML diagram for the class.

Figure 4-15 UML diagram for the SalesCommission class



[Figure 4-15 Full Alternative Text](#)

[Table 4-12](#) lists and describes the class's fields.

Table 4-12 SalesCommission class fields

Field	Description
sales	A double variable to hold a salesperson's total monthly sales.
rate	A double variable to hold the salesperson's commission rate.
commission	A double variable to hold the commission.
advance	A double variable to hold the amount of advanced pay the salesperson has drawn.
pay	A double variable to hold the salesperson's amount of gross pay.

[Table 4-13](#) lists and describes the class's methods.

Table 4-13 SalesCommission class methods

Method	Description
Constructor	The constructor accepts two arguments: the amount of sales that a salesperson has made, and the amount of advanced pay that salesperson has drawn. The method assigns these values to the sales and advance fields, then calls the calculatePay method.
setRate	A private method that sets the rate of commission, based on the amount of sales made by the

	salesperson. This method is called from the calculatePay method.
calculatePay	A private method that calculates the salesperson's commission and actual pay. This method is called from the constructor.
getPay	Returns as a double the amount of gross pay due the salesperson, which is the amount of commission minus advanced pay.
getComission	Returns as a double the amount of commission earned by the salesperson.
getRate	Returns as a double the rate of commission for the amount of sales made by the salesperson.
getAdvance	Returns as a double the amount of advanced pay drawn by the salesperson.
getSales	Returns as a double the amount of sales made by the salesperson.

[Code Listing 4-19](#) shows the code for the class.

Code Listing 4-19 (SalesCommission.java)

```

1  /**
2   * This class calculates a salesperson's gross
3   * pay based on the amount of sales.
4  */
5
6 public class SalesCommission
7 {
8     private double sales,           // Monthly sales
9                     rate,          // Rate of commission
10                commission, // Amount of commission
11                advance,    // Advanced pay
12                pay;         // Amount to pay
13
14
15 /**

```

```
16     * The constructor uses two parameters to accept
17     * arguments: s and a. The value in s is assigned to
18     * the sales field and the value in a is assigned to
19     * the advance field. The calculatePay method is called.
20     */
21
22     public SalesCommission(double s, double a)
23     {
24         sales = s;
25         advance = a;
26         calculatePay();
27     }
28
29     /**
30      * The setRate method sets the rate of commission,
31      * based on the amount of sales. This method is called
32      * from the calculatePay method.
33      */
34
35     private void setRate()
36     {
37         if (sales < 10000)
38             rate = 0.05;
39         else if (sales < 15000)
40             rate = 0.1;
41         else if (sales < 18000)
42             rate = 0.12;
43         else if (sales < 22000)
44             rate = 0.14;
45         else
46             rate = 0.16;
47     }
48
49     /**
50      * The calculatePay method calculates the salesperson's
51      * commission and amount of actual pay.
52      */
53
54     private void calculatePay()
55     {
56         setRate();
57         commission = sales * rate;
58         pay = commission - advance;
59     }
60
61     /**
62      * The getPay method returns the pay field.
63      */
```

```
64     public double getPay()
65     {
66         return pay;
67     }
68
69
70     /**
71      * The getCommission method returns the commission field.
72      */
73
74     public double getCommission()
75     {
76         return commission;
77     }
78
79     /**
80      * The getRate method returns the rate field.
81      */
82
83     public double getRate()
84     {
85         return rate;
86     }
87
88     /**
89      * The getAdvance method returns the advance field.
90      */
91
92     public double getAdvance()
93     {
94         return advance;
95     }
96
97     /**
98      * The getSales method returns the sales field.
99      */
100
101    public double getSales()
102    {
103        return sales;
104    }
105 }
```

Private Methods and Algorithm

Decomposition

Notice that the class has two private methods: `setRate` and `calculatePay`. When a method is declared as `private`, it can be called only from other methods that are members of the same class. Sometimes, a class will contain methods that are necessary for internal processing, but are not useful to code outside of the class. These methods are usually declared as `private`.

In the case of the `SalesCommission` class, the `setRate` and `calculatePay` methods are part of an algorithm that has been *decomposed*. Decomposing an algorithm usually means breaking it into several short methods, each performing a specific task. For example, look at the `SalesCommission` class's constructor, which appears in lines 22 through 27. Notice that in line 26, the constructor calls the `calculatePay` method. Then, in line 56 the `calculatePay` method calls the `setRate` method, which appears in lines 35 through 47.

All three of these methods, the constructor, `calculatePay`, and `setRate`, form the pieces of a single algorithm. The entire algorithm could have been written in the constructor, which would then look something like this:

```
// The entire algorithm written in the constructor!
public void SalesCommission(double s, double a)
{
    sales = s;
    advance = a;
    if (sales < 10000)
        rate = 0.05;
    else if (sales < 15000)
        rate = 0.1;
    else if (sales < 18000)
        rate = 0.12;
    else if (sales < 22000)
        rate = 0.14;
    else
        rate = 0.16;
    commission = sales * rate;
    pay = commission - advance;
}
```

Can you see how decomposing this algorithm into three methods improves the code? First, it isolates the related tasks into separate methods: The constructor assigns initial values to the sales and advance fields; the setRate method determines the commission rate; and the calculatePay method calculates the salesperson's gross pay. Second, code that is broken up into small, related chunks is easier to read and debug than one long method that performs many tasks.

The Main Program

The main program code is shown in [Code Listing 4-20](#). First, it gets the amount of sales and advanced pay for a salesperson as input from the user. It then creates an instance of the SalesCommission class and passes this data to the class's constructor. The program then reads the resultant pay data from the SalesCommission object and displays it on the screen.

Code Listing 4-20 (HalsCommission.java)

```

20     System.out.println("Enter the following information:");
21
22     // Ask the user for sales & Advanced Pay
23     System.out.print("Amount of sales: $");
24     sales = keyboard.nextDouble();
25     System.out.print("Amount of advanced pay: $");
26     advancePay = keyboard.nextDouble();
27
28     // Create an instance of the SalesCommission
29     // class and pass the data to the constructor.
30     SalesCommission payInfo =
31         new SalesCommission(sales, advancePay);
32
33     // Display the pay report for the salesperson.
34     System.out.println("\nPay Report");
35     System.out.println("-----");
36     System.out.printf("Sales: $%,.2f\n", payInfo.getSales());
37     System.out.printf("Commission rate: %.2f\n", payInfo.getCommissionRate());
38     System.out.printf("Commission: $%,.2f\n", payInfo.getCommission());
39     System.out.printf("Advanced pay: $%,.2f\n", payInfo.getAdvancedPay());
40     System.out.printf("Remaining pay: $%,.2f\n", payInfo.getRemainingPay());
41 }
42 }
```

Program Output with Example Input Shown in Bold

This program will display a pay report for a salesperson.
Enter the following information:

Amount of sales: **\$19600** 

Amount of advanced pay: **\$1000** 

Pay Report

 Sales: \$19,600.00
 Commission rate: 0.14
 Commission: \$2,744.00
 Advanced pay: \$1,000.00
 Remaining pay: \$1,744.00

4.12 Generating Random Numbers with the Random Class

Concept:

Random numbers are used in a variety of applications. Java provides the `Random` class, which you can use to generate random numbers.

Random numbers are useful for lots of different programming tasks. The following are just a few examples:

- Random numbers are commonly used in games; for example, computer games that let the player roll dice use random numbers to represent the values of the dice. Programs that show cards being drawn from a shuffled deck use random numbers to represent the face values of the cards.
- Random numbers are useful in simulation programs. In some simulations, the computer must randomly decide how a person, animal, insect, or other living being will behave. Formulas can be constructed in which a random number is used to determine various actions and events that take place in the program.
- Random numbers are useful in statistical programs that must randomly select data for analysis.
- Random numbers are commonly used in computer security to encrypt sensitive data.

The Java API provides a class named `Random` that you can use to generate random numbers. The class is part of the `java.util` package, so any program

that uses it will need an `import` statement such as

```
import java.util.Random;
```

You create an object from the `Random` class with a statement such as this:

```
Random randomNumbers = new Random();
```

This statement does the following:

- It declares a variable named `randomNumbers`. The data type is the `Random` class.
- The expression `new Random()` creates an instance of the `Random` class.
- The equal sign assigns the address of the `Random` class to the `randomNumbers` variable.

After this statement executes, the `randomNumbers` variable references a `Random` object. Once you have created a `Random` object, you can call its `nextInt` method to get a random integer number. The following code shows an example:

```
// Declare an int variable.  
int number;  
  
// Create a Random object.  
Random randomNumbers = new Random();  
  
// Get a random integer and assign it to number.  
number = randomNumbers.nextInt();
```

After this code executes, the `number` variable contains a random integer. If you call the `nextInt` method with no arguments, as shown in this example, the returned integer is somewhere between $-2,147,483,648$ and $+2,147,483,647$. Alternatively, you can pass an argument that specifies an upper limit to the generated number's range. In the following statement, the value assigned to `number` is somewhere between 0 and 99:

```
number = randomNumbers.nextInt(100);
```

You can add or subtract a value to shift the numeric range upward or downward. In the following statement, we call the `nextInt` method to get a random number in the range of 0 through 9, and then we add 1 to it. So, the number assigned to `number` is somewhere in the range of 1 through 10:

```
number = randomNumbers.nextInt(10) + 1;
```

The following statement shows another example. It assigns a random integer between -50 and 49 to `number`:

```
number = randomNumbers.nextInt(100) - 50;
```

The `Random` class has other methods for generating random numbers. [Table 4-14](#) summarizes several of them.

Table 4-14 Some of the Random class's methods

Method	Description
<code>nextDouble()</code>	Returns the next random number as a <code>double</code> . The number will be within the range of <code>0.0</code> and <code>1.0</code> .
<code>nextFloat()</code>	Returns the next random number as a <code>float</code> . The number will be within the range of <code>0.0</code> and <code>1.0</code> .
<code>nextInt()</code>	Returns the next random number as an <code>int</code> . The number will be within the range of an <code>int</code> , which is <code>-2,147,483,648</code> to <code>+2,147,483,647</code> .
<code>nextInt(int n)</code>	This method accepts an integer argument, <code>n</code> . It returns a random number as an <code>int</code> . The number will be within the range of <code>0</code> to <code>n</code> , not including <code>n</code> .
<code>nextLong()</code>	Returns the next random number as a <code>long</code> . The number will be within the range of a <code>long</code> , which is <code>-9,223,372,036,854,775,808</code> to <code>+9,223,372,036,854,775,807</code> .

The program in [Code Listing 4-21](#) demonstrates the Random class.

Code Listing 4-21 (MathTutor.java)

```
1 import java.util.Scanner; // Needed for Scanner class
2 import java.util.Random; // Needed for Random class
3
4 /**
5  * This program demonstrates the Random class.
6 */
7
8 public class MathTutor
9 {
10     public static void main(String[] args)
11     {
12         int number1;      // First number
13         int number2;      // Second number
14         int sum;          // Sum of numbers
15         int userAnswer;   // User's answer
16
17         // Create a Scanner object for keyboard input.
18         Scanner keyboard = new Scanner(System.in);
19
20         // Create a Random object.
21         Random randomNumbers = new Random();
22
23         // Get two random numbers.
24         number1 = randomNumbers.nextInt(100);
25         number2 = randomNumbers.nextInt(100);
26
27         // Display an addition problem.
28         System.out.println("What is the answer to " +
29                             "the following problem?");
30         System.out.print(number1 + " + " +
31                         number2 + " = ? ");
32
33         // Calculate the answer.
34         sum = number1 + number2;
35
36         // Get the user's answer.
37         userAnswer = keyboard.nextInt();
38
39         // Display the user's results.
40         if (userAnswer == sum)
41             System.out.println("Correct!");
```

```
42     else
43     {
44         System.out.println("Sorry, wrong answer. " +
45                         "The correct answer is " +
46                         sum);
47     }
48 }
49 }
```

Program Output with Example Input Shown in Bold

What is the answer to the following problem?

$52 + 19 = ?$ **71** 

Correct!

Program Output with Example Input Shown in Bold

What is the answer to the following problem?

$27 + 73 = ?$ **101** 

Sorry, wrong answer. The correct answer is 100

In the Spotlight: Simulating Dice

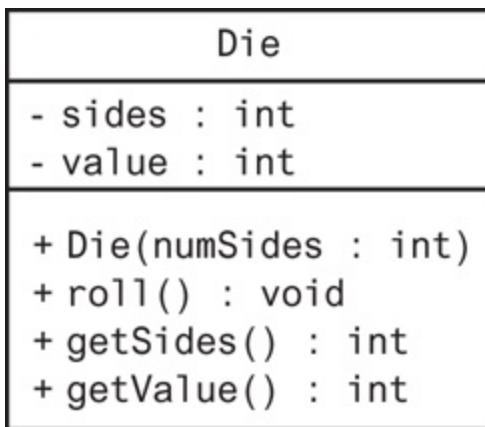
with Objects



Dice traditionally have six sides, representing the values 1 to 6. Some games, however, use specialized dice that have different numbers of sides. For example, the fantasy role-playing game *Dungeons and Dragons*® uses dice with 4, 6, 8, 10, 12, and 20 sides.

Suppose you are writing a program that needs to roll simulated dice with various numbers of sides. A simple approach would be to write a `Die` class with a constructor that accepts the number of sides as an argument. The class would also have appropriate methods for rolling the die, and getting the die's value. [Figure 4-16](#) shows the UML diagram for such a class, and [Code Listing 4-22](#) shows the code.

Figure 4-16 UML diagram for the Die class



[Figure 4-16 Full Alternative Text](#)

Code Listing 4-22 (Die.java)

```
1 import java.util.Random;
2
3 /**
4  * The Die class simulates a die with a specified number of
5  */
6
7 public class Die
8 {
9     private int sides;    // Number of sides
10    private int value;   // The die's value
11
12 /**
13  * The constructor performs an initial
14  * roll of the die. The number of sides
15  * for the die is passed as an argument.
16  */
17
18 public Die(int numSides)
19 {
20     sides = numSides;
21     roll();
```

```

22     }
23
24     /**
25      * The roll method simulates the rolling of
26      * the die.
27     */
28
29     public void roll()
30     {
31         // Create a Random object.
32         Random rand = new Random();
33
34         // Get a random value for the die.
35         value = rand.nextInt(sides) + 1;
36     }
37
38     /**
39      * The getSides method returns the
40      * number of sides for the die.
41     */
42
43     public int getSides()
44     {
45         return sides;
46     }
47
48     /**
49      * The getValue method returns the
50      * value of the die.
51     */
52
53     public int getValue()
54     {
55         return value;
56     }
57 }
```

Let's take a closer look at the code for the class:

- Lines 9 and 10: These statements declare two `int` fields. The `sides` field will hold the number of sides that the die has, and the `value` field will hold the value of the die once it has been rolled.
- Lines 18–22: This is the constructor. Notice that the constructor has a parameter for the number of sides. The parameter is assigned to the

sides field in line 20. Line 21 calls the roll method, which simulates the rolling of the die.

- Lines 29–36: This is the roll method, which simulates the rolling of the die. In line 32, a Random object is created, and it is referenced by the rand variable. Line 35 uses the Random object to get a random number that is in the appropriate range for this particular die. For example, if the sides field is set to 6, the expression rand.nextInt(sides) + 1 will return a random integer in the range of 1 through 6. The random number is assigned to the value field.
- Lines 43–46: This is the getSides method, an accessor that returns the sides field.
- Lines 53–56: This is the getValue method, an accessor that returns the value field.

The program in [Code Listing 4-23](#) demonstrates the class. It creates two instances of the Die class: one with 6 sides, and the other with 12 sides. It then simulates rolling the dice.

Code Listing 4-23 (DiceDemo.java)

```
1  /**
2   * This program simulates the rolling of dice.
3   */
4
5 public class DiceDemo
6 {
7     public static void main(String[] args)
8     {
9         final int DIE1_SIDES = 6; // Number of sides for die #
10        final int DIE2_SIDES = 12; // Number of sides for die #
11
12        // Create two instances of the Die class.
13        Die die1 = new Die(DIE1_SIDES);
14        Die die2 = new Die(DIE2_SIDES);
15
16        // Display initial information.
17        System.out.println("This simulates the rolling of a " +
```

```

18                     DIE1_SIDES + " sided die and a " +
19                     DIE2_SIDES + " sided die.");
20
21     // Roll the dice.
22     System.out.println("Rolling the dice.");
23     die1.roll();
24     die2.roll();
25
26     // Display the values of the dice.
27     System.out.println(die1.getValue() + " " +
28                         die2.getValue());
29 }
30 }
```

Program Output

This simulates the rolling of a 6 sided die and a 12 sided die.
 Rolling the dice.
 5 10

Let's take a closer look at the program:

- Lines 9–10: These statements declare two constants. `DIE1_SIDES` is the number of sides for the first die (6), and `DIE2_SIDES` is the number of sides for the second die (12).
- Lines 13–14: These statements create two instances of the `Die` class. Notice that `DIE1_SIDES`, which is 6, is passed to the constructor in line 13, and `DIE2_SIDES`, which is 12, is passed to the constructor in line 14. As a result, `die1` will reference a `Die` object with 6 sides, and `die2` will reference a `Die` object with 12 sides.
- Lines 17–19: This statement displays information about the program.
- Lines 23–24: These statements call the `Die` objects' `roll` method to simulate rolling the dice.
- Lines 27–28: This statement displays the value of the dice.

4.13 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- Using `=` instead of `==` to compare primitive values. Remember, `=` is the assignment operator and `==` tests for equality.
- Using `==` instead of the `equals` method to compare `String` objects. You cannot use the `==` operator to compare the contents of a `String` object with another string. Instead, you must use the `equals` or `compareTo` methods.
- Forgetting to enclose an `if` statement's boolean expression in parentheses. Java requires that the boolean expression being tested by an `if` statement be enclosed in a set of parentheses. An error will result if you omit the parentheses or use any other grouping characters.
- Writing a semicolon at the end of an `if` clause. When you write a semicolon at the end of an `if` clause, Java assumes that the conditionally executed statement is a null or empty statement.
- Forgetting to enclose multiple conditionally executed statements in braces. Normally the `if` statement conditionally executes only one statement. To conditionally execute more than one statement, you must enclose them in braces.
- Omitting the trailing `else` in an `if-else-if` statement. This is not a syntax error, but can lead to logical errors. If you omit the trailing `else` from an `if-else-if` statement, no code will be executed if none of the statement's boolean expressions are `true`.
- Not writing complete boolean expressions on both sides of a logical `&&` or `||` operator. You must write a complete boolean expression on both sides of a logical `&&` or `||` operator. For example, the expression `x > 0 && < 10` is not valid because `< 10` is not a complete expression. The

expression should be written as `x > 0 && x < 10`.

- Trying to perform case-insensitive string comparisons with the `String` class's `equals` and `compareTo` methods. To perform case-insensitive string comparisons, use the `String` class's `equalsIgnoreCase` and `compareToIgnoreCase` methods.
- Using an invalid `testExpression` in a `switch` statement. The `switch` statement can evaluate `char`, `byte`, `short`, `int`, or `String` expressions.
- Forgetting to write a colon at the end of a `case` statement. A colon must appear after the `CaseExpression` in each `case` statement.
- Forgetting to write a `break` statement in a `case` section. This is not a syntax error, but it can lead to logical errors. The program does not branch out of a `switch` statement until it reaches a `break` statement or the end of the `switch` statement.
- Forgetting to write a `default` section in a `switch` statement. This is not a syntax error, but can lead to a logical error. If you omit the `default` section, no code will be executed if none of the `case` values match the `testExpression`.
- Reversing the `?` and the `:` when using the conditional operator. When using the conditional operator, the `?` character appears first in the conditional expression, then the `:` character.

Review Questions and Exercises

Multiple Choice and True/False

1. The `if` statement is an example of a .
 1. sequence structure.
 2. decision structure.
 3. pathway structure.
 4. class structure.

2. This type of expression has a value of either `true` or `false`.
 1. binary expression
 2. decision expression
 3. unconditional expression
 4. boolean expression

3. `>`, `<`, and `==` are .
 1. relational operators.
 2. logical operators.
 3. conditional operators.
 4. ternary operators.

4. `&&`, `||`, and `!` are .

1. relational operators.
 2. logical operators.
 3. conditional operators.
 4. ternary operators.
5. This is an empty statement that does nothing.
1. missing statement
 2. virtual statement
 3. null statement
 4. conditional statement
6. To create a block of statements, you enclose the statements in .
1. parentheses ()
 2. square brackets []
 3. angled brackets <>
 4. braces {}
7. This is a boolean variable that signals when some condition exists in the program.
1. flag
 2. signal
 3. sentinel
 4. siren

8. How does the character “A” compare to the character “B”?
 1. “A” is greater than “B”
 2. “A” is less than “B”
 3. “A” is equal to “B”
 4. You cannot compare characters.
9. This is an `if` statement that appears inside another `if` statement.
 1. nested `if` statement
 2. tiered `if` statement
 3. dislodged `if` statement
 4. structured `if` statement
10. An `else` clause always goes with which of the following?
 1. the closest previous `if` clause that doesn’t already have its own `else` clause.
 2. the closest `if` clause.
 3. the `if` clause that is randomly selected by the compiler.
 4. none of these
11. When determining whether a number is inside a range, it’s best to use this operator.
 1. `&&`
 2. `!`
 3. `||`

4. ? :

12. This determines whether two different `String` objects contain the same string.

1. the `==` operator
2. the `=` operator
3. the `equals` method
4. the `stringCompare` method

13. The conditional operator takes how many operands?

1. one
2. two
3. three
4. four

14. This section of a `switch` statement is branched to if none of the case values match the *testExpression*.

1. `else`
2. `default`
3. `case`
4. `otherwise`

15. **True or False:** The `=` operator and the `==` operator perform the same operation.

16. **True or False:** A conditionally executed statement should be indented one level from the `if` clause.

17. **True or False:** All lines in a conditionally executed block should be indented one level.
18. **True or False:** When an `if` statement is nested in the `if` clause of another statement, the only time the inner `if` statement is executed is when the boolean expression of the outer `if` statement is `true`.
19. **True or False:** When an `if` statement is nested in the `else` clause of another statement, the only time the inner `if` statement is executed is when the boolean expression of the outer `if` statement is `true`.
20. **True or False:** The scope of a variable is limited to the block in which it is defined.

Find the Error

Find the errors in the following code:

1. // Warning! This code contains ERRORS!

```
if (x == 1);
    y = 2;
else if (x == 2);
    y = 3;
else if (x == 3);
    y = 4;
```
2. // Warning! This code contains an ERROR!

```
if (average = 100)
    System.out.println("Perfect Average!");
```
3. // Warning! This code contains ERRORS!

```
if (num2 == 0)
    System.out.println("Division by zero is not possible.");
    System.out.println("Please run the program again ");
    System.out.println("and enter a number besides zero.");
else
    Quotient = num1 / num2;
    System.out.print("The quotient of " + Num1);
    System.out.print(" divided by " + Num2 + " is ");
    System.out.println(Quotient);
```

4. // Warning! This code contains ERRORS!

```
switch (score)
{
    case (score > 90):
        grade = 'A';
        break;
    case(score > 80):
        grade = 'b';
        break;
    case(score > 70):
        grade = 'C';
        break;
    case (score > 60):
        grade = 'D';
        break;
    default:
        grade = 'F';
}
```

5. The following statement should determine whether *x* is not greater than 20. What is wrong with it?

```
if (!x > 20)
```

6. The following statement should determine whether *count* is within the range of 0 through 100. What is wrong with it?

```
if (count >= 0 || count <= 100)
```

7. The following statement should determine whether *count* is outside the range of 0 through 100. What is wrong with it?

```
if (count < 0 && count > 100)
```

8. The following statement should assign 0 to *z* if *a* is less than 10; otherwise it should assign 7 to *z*. What is wrong with it?

```
z = (a < 10) : 0 ? 7;
```

9. Assume that *partNumber* references a *String* object. The following *if* statement should perform a case-insensitive comparison. What is wrong with the code?

```
if (partNumber.equals("BQ789W4"))
    available = true;
```

Algorithm Workbench

1. Write an `if` statement that assigns 100 to `x` when `y` is equal to 0.
2. Write an `if-else` statement that assigns 0 to `x` when `y` is equal to 10. Otherwise, it should assign 1 to `x`.
3. Using the following chart, write an `if-else-if` statement that assigns .10, .15, or .20 to `commission`, depending on the value in `sales`:

Sales	Commission Rate
Up to \$10,000	10%
\$10,000 to \$15,000	15%
Over \$15,000	20%

4. Write an `if` statement that sets the variable `hours` to 10 when the flag variable `minimum` is equal to `true`.
5. Write nested `if` statements that perform the following tests: If `amount1` is greater than 10, and `amount2` is less than 100, display the greater of the two.
6. Write an `if` statement that prints the message “The number is valid” if the variable `grade` is within the range 0 through 100.
7. Write an `if` statement that prints the message “The number is valid” if the variable `temperature` is within the range -50 through 150.
8. Write an `if` statement that prints the message “The number is not valid” if the variable `hours` is outside the range 0 through 80.
9. Write an `if-else` statement that displays the `String` objects `title1` and `title2` in alphabetical order.

10. Convert the following if-else-if statement into a switch statement:

```
if (choice == 1)
{
    System.out.println("You selected 1.");
}
else if (choice == 2 || choice == 3)
{
    System.out.println("You selected 2 or 3.");
}
else if (choice == 4)
{
    System.out.println("You selected 4.");
}
else
{
    System.out.println("Select again please.");
}
```

11. Match the conditional expression with the if-else statement that performs the same operation.

1. $q = x < y ? a + b : x * 2;$

2. $q = x < y ? x * 2 : a + b;$

3. $x < y ? q = 0 : q = 1;$

_____ if ($x < y$)
 q = 0;
 else
 q = 1;
_____ if ($x < y$)
 q = a + b;
 else
 q = $x * 2$;
_____ if ($x < y$)
 q = $x * 2$;
 else
 q = a + b;

Short Answer

1. Explain what is meant by the term “conditionally executed.”
2. Explain why a misplaced semicolon can cause an `if` statement to operate incorrectly.
3. Why is it good advice to indent all the statements inside a set of braces?
4. What happens when you compare two `String` objects with the `==` operator?
5. Explain the purpose of a flag variable. Of what data type should a flag variable be?
6. What risk does a programmer take when not placing a trailing `else` at the end of an `if-else-if` statement?
7. Briefly describe how the `&&` operator works.
8. Briefly describe how the `||` operator works.
9. Why are the relational operators called “relational”?
10. How do you use `private` methods in a class to decompose an algorithm?

Programming Challenges

1. Roman Numerals

Write a program that prompts the user to enter a number within the range of 1 through 10. The program should display the Roman numeral version of that number. If the number is outside the range of 1–10, the program should display an error message.



VideoNote The Time Calculator Problem

2. Time Calculator

Write a program that asks the user to enter a number of seconds.

- There are 60 seconds in a minute. If the number of seconds entered by the user is greater than or equal to 60, the program should display the number of minutes and leftover seconds in that many seconds.
- There are 3,600 seconds in an hour. If the number of seconds entered by the user is greater than or equal to 3,600, the program should display the number of hours, minutes, and leftover seconds in that many seconds.
- There are 86,400 seconds in a day. If the number of seconds entered by the user is greater than or equal to 86,400, the program should display the number of days, hours, minutes, and leftover seconds in that many seconds.

3. TestScores Class

Design a `TestScores` class that has fields to hold three test scores. (If you have already written the `TestScores` class for Programming

Challenge 8 of [Chapter 3](#), you can modify it.) The class constructor should accept three test scores as arguments and assign these arguments to the test score fields. The class should also have accessor methods for the test score fields, a method that returns the average of the test scores, and a method that returns the letter grade that is assigned for the test score average. Use the grading scheme in the following table:

Test Score Average Letter Grade

90–100	A
80–89	B
70–79	C
60–69	D
Below 60	F

4. Software Sales

A software company sells a package that retails for \$99. Quantity discounts are given according to the following table:

Quantity	Discount
10–19	20%
20–49	30%
50–99	40%
100 or more	50%

Design a class that stores the number of units sold, and has a method that returns the total cost of the purchase.

5. BankCharges Class

A bank charges \$10 per month, plus the following check fees for a commercial checking account:

- \$.10 for each check if less than 20 checks were written
- \$.08 for each check if 20 through 39 checks were written

- \$.06 for each check if 40 through 59 checks were written
- \$.04 for each check if 60 or more checks were written

The bank also charges an extra \$15 if the account balance falls below \$400 (before any check fees are applied). Design a class that stores the ending balance of an account and the number of checks written. It should also have a method that returns the bank's service fees for the month.

6. ShippingCharges Class

The Fast Freight Shipping Company charges the following rates:

Weight of Package (in kilograms) Rate per 500 Miles Shipped

2 kg or less	\$1.10
Over 2 kg but not more than 6 kg	\$2.20
Over 6 kg but not more than 10 kg	\$3.70
Over 10 kg	\$4.80

The shipping charges per 500 miles are not prorated. For example, if a 2 kg package is shipped 550 miles, the charges would be \$2.20.

Design a class that stores the weight of a package, and has a method that returns the shipping charges.

7. FatGram Class

Design a class with a method that stores the number of calories and fat grams in a food item. The class should have a method that returns the percentage of the calories that come from fat. One gram of fat has 9 calories, so:

$$\text{Calories from fat} = \text{fat grams} * 9$$

The percentage of calories from fat can be calculated as:

$$\text{Calories from fat} \div \text{total calories}$$

Demonstrate the class in a program that asks the user to enter the number of calories and the number of fat grams for a food item. The program should display the percentage of calories that come from fat. If the calories from fat are less than 30 percent of the total calories of the food, it should also display a message indicating the food is low in fat.

Because the number of calories from fat cannot be greater than the total number of calories, if the user enters a number for the calories from fat that is greater than the total number of calories, the program should display an error message indicating that the numbers are invalid.

8. Running the Race

Design a class that stores the names of three runners and the time, in minutes, it took each of them to finish a race. The class should have methods that return the name of the runner in 1st, 2nd, or 3rd place.

9. The Speed of Sound

The following table shows the approximate speed of sound in air, water, and steel:

Medium	Speed
Air	1,100 feet per second
Water	4,900 feet per second
Steel	16,400 feet per second

Design a class that stores in a *distance* field the distance, in feet, traveled by a sound wave. The class should have the appropriate accessor and mutator methods for this field. In addition, the class should have the following methods:

- `getSpeedInAir`. This method should return the number of seconds it would take a sound wave to travel, in air, the distance stored in the *distance* field. The formula to calculate the amount of time it will take the sound wave to travel the specified distance in air is:

$$\text{Time} = \frac{\text{distance}}{1100}$$

- `getSpeedInWater`. This method should return the number of seconds it would take a sound wave to travel, in water, the distance stored in the `distance` field. The formula to calculate the amount of time it will take the sound wave to travel the specified distance in water is:

`Time 5 distance/4900`

- `getSpeedInSteel`. This method should return the number of seconds it would take a sound wave to travel, in steel, the distance stored in the `distance` field. The formula to calculate the amount of time it will take the sound wave to travel the specified distance in air is:

`Time 5 distance/16400`

Write a program to demonstrate the class. The program should display a menu allowing the user to select air, water, or steel. Once the user has made a selection, he or she should be asked to enter the distance a sound wave will travel in the selected medium. The program will then display the amount of time it will take. Check that the user has selected one of the available choices from the menu.

10. Freezing and Boiling Points

The following table lists the freezing and boiling points of several substances in Fahrenheit:

Substance	Freezing Point	Boiling Point
Ethyl alcohol	-173	172
Oxygen	-362	-306
Water	32	212

Design a class that stores a temperature in a `temperature` field and has the appropriate accessor and mutator methods for the field. The class should also have the following methods:

- `isEthylFreezing`. This method should return the boolean value

true if the temperature stored in the temperature field is at or below the freezing point of ethyl alcohol. Otherwise, the method should return false.

- `isEthylBoiling`. This method should return the boolean value true if the temperature stored in the temperature field is at or above the boiling point of ethyl alcohol. Otherwise, the method should return false.
- `isOxygenFreezing`. This method should return the boolean value true if the temperature stored in the temperature field is at or below the freezing point of oxygen. Otherwise, the method should return false.
- `isOxygenBoiling`. This method should return the boolean value true if the temperature stored in the temperature field is at or above the boiling point of oxygen. Otherwise, the method should return false.
- `isWaterFreezing`. This method should return the boolean value true if the temperature stored in the temperature field is at or below the freezing point of water. Otherwise, the method should return false.
- `isWaterBoiling`. This method should return the boolean value true if the temperature stored in the temperature field is at or above the boiling point of water. Otherwise, the method should return false.

Write a program that demonstrates the class. The program should ask the user to enter a temperature, then display a list of the substances that will freeze at that temperature and those that will boil at that temperature. For example, if the temperature is -20, the class should report that water will freeze and oxygen will boil at that temperature.

11. Mobile Service Provider

A mobile phone service provider has three different subscription

packages for its customers:

Package For \$39.99 per month, 450 minutes are provided. Additional A: minutes are \$0.45 per minute.

Package For \$59.99 per month, 900 minutes are provided. Additional B: minutes are \$0.40 per minute.

Package C For \$69.99 per month, unlimited minutes are provided.

Design a class that calculates a customer's monthly bill. It should store the letter of the package the customer has purchased (A, B, or C) and the number of minutes that were used. It should have a method that returns the total charges. Demonstrate the class in a program that asks the user to select a package and enter the number of minutes used. The program should display the total charges.

12. Mobile Service Provider, Part 2

Modify the program you wrote for Programming Challenge 11 so it also calculates and displays the amount of money Package A customers would save if they purchased package B or C, and the amount of money package B customers would save if they purchased package C. If there would be no savings, no message should be printed.

13. Body Mass Index

Write a program that calculates and displays a person's body mass index (BMI). The BMI is often used to determine whether a person is overweight or underweight for his or her height. A person's BMI is calculated with the following formula:

$$\text{BMI} = \text{weight} \times 703/\text{height}^2$$

where *weight* is measured in pounds, and *height* is measured in inches. The program should display a message indicating whether the person has optimal weight, is underweight, or is overweight. A person's weight is considered to be optimal if his or her BMI is between 18.5 and 25. If the BMI is less than 18.5, the person is considered to be underweight. If

the BMI value is greater than 25, the person is considered to be overweight.

14. Days in a Month

Write a class named `MonthDays`. The class's constructor should accept two arguments:

- An integer for the month (1 = January, 2 = February, etc.)
- An integer for the year

The class should have a method named `getNumberOfDays` that returns the number of days in the specified month. The method should use the following criteria to identify leap years:

1. Determine whether the year is evenly divisible by 100. If it is, then it is a leap year if and only if it is evenly divisible by 400. For example, 2000 is a leap year, but 2100 is not.
2. If the year is not evenly divisible by 100, then it is a leap year if and only if it is evenly divisible by 4. For example, 2008 is a leap year but 2009 is not.

Demonstrate the class in a program that asks the user to enter the month (letting the user enter an integer in the range of 1 through 12) and the year. The program should then display the number of days in that month. Here is a sample run of the program:

```
Enter a month (1-12): 2 Enter
Enter a year: 2020 Enter
29 days
```

15. Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month, then displays the number of points awarded.

16. Magic Dates

The date June 10, 1960, is special because when we write it in the following format, the month times the day equals the year.

6/10/60

Design a class named `MagicDate`. The class constructor should accept, as integers, values for a month, a day, and a year. The class should also have a method named `isMagic` that returns `true` if the date passed to the constructor is magic, or `false` otherwise.

Write a program that asks the user to enter a month, a day, and a two-digit year as integers. The program should create an instance of the `MagicDate` class to determine whether the date entered by the user is a magic date. If it is, the program should display a message saying the date is magic. Otherwise, it should display a message saying the date is not magic.

17. Hot Dog Cookout Calculator

Assume that hot dogs come in packages of 10, and hot dog buns come in packages of 8. Write a program that calculates the number of packages of hot dogs and the number of packages of hot dog buns needed for a cookout, with the minimum amount of leftovers. The program should

ask the user for the number of people attending the cookout, and the number of hot dogs each person will be given. The program should display:

- The minimum number of packages of hot dogs required.
- The minimum number of packages of buns required.
- The number of hot dogs that will be left over.
- The number of buns that will be left over.

18. Roulette Wheel Colors

On a roulette wheel, the pockets are numbered from 0 to 36, with an additional pocket numbered 00. The colors of the pockets are as follows:

- Pockets 0 and 00 are green.
- For pockets 1 through 10, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 11 through 18, the odd numbered pockets are black and the even numbered pockets are red.
- For pockets 19 through 28, the odd numbered pockets are red and the even numbered pockets are black.
- For pockets 29 through 36, the odd numbered pockets are black and the even numbered pockets are red.

Write a class named `RoulettePocket`. The class's constructor should accept a pocket number. The class should have a method named `getPocketColor` that returns the pocket's color, as a string.

Demonstrate the class in a program that asks the user to enter a pocket number, and displays whether the pocket is green, red, or black. The program should display an error message if the user enters a number that is outside the range of 0 through 36.

19. Wi-Fi Diagnostic Tree

[Figure 4-17](#) shows a simplified flowchart for troubleshooting a bad Wi-Fi connection. Use the flowchart to create a program that leads a person through the steps of fixing a bad Wi-Fi connection. Here is an example of the program's output:

Reboot the computer and try to connect.

Did that fix the problem? no

Reboot the router and try to connect.

Did that fix the problem? yes

Figure 4-17 Troubleshooting a bad Wi-Fi connection

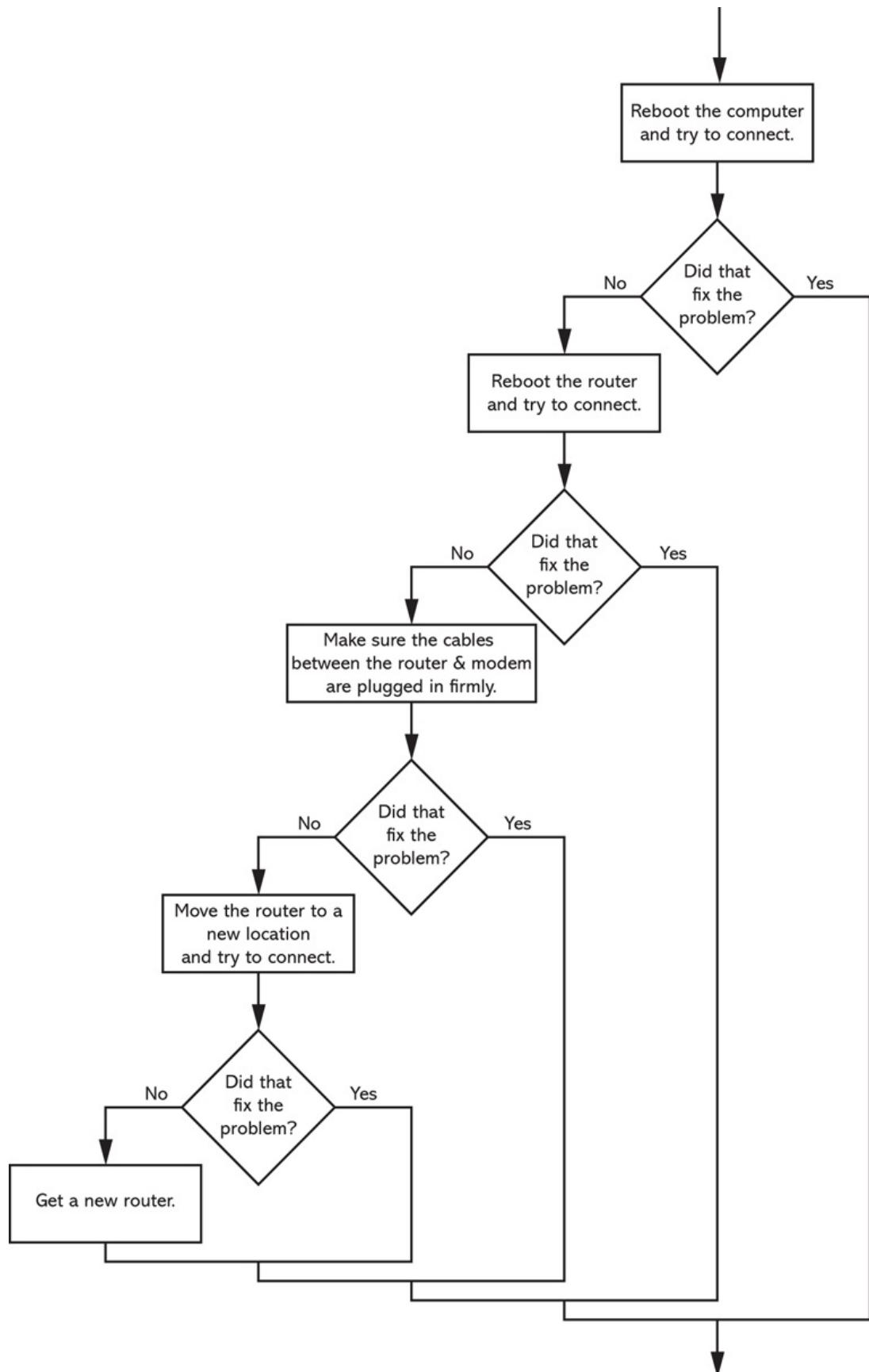


Figure 4-17 Full Alternative Text

Notice that the program ends as soon as a solution is found to the problem. Here is another example of the program's output:

Reboot the computer and try to connect.

Did that fix the problem? no

Reboot the router and try to connect.

Did that fix the problem? no

Make sure the cables between the router & modem are plugged in.

Did that fix the problem? no

Move the router to a new location.

Did that fix the problem? no

Get a new router.

20. Restaurant Selector

You have a group of friends coming to visit for your high school reunion, and you want to take them out to eat at a local restaurant. You aren't sure if any of them have dietary restrictions, but your restaurant choices are as follows:

Restaurant	Vegetarian	Vegan	Gluten-Free
Joe's Gourmet Burgers	No	No	No
Main Street Pizza Company	Yes	No	Yes
Corner Café	Yes	Yes	Yes
Mama's Fine Italian	Yes	No	No
The Chef's Kitchen	Yes	Yes	Yes

Write a program that asks whether any members of your party are vegetarian, vegan, or gluten-free, then displays only the restaurants where you may take the group. Here is an example of the program's output:

Is anyone in your party a vegetarian? yes

Is anyone in your party a vegan? no

Is anyone in your party gluten-free? yes

Here are your restaurant choices:

Main Street Pizza Company

Corner Cafe

The Chef's Kitchen

Here is another example of the program's output:

Is anyone in your party a vegetarian? yes

Is anyone in your party a vegan? yes

Is anyone in your party gluten-free? yes

Here are your restaurant choices:

Corner Cafe

The Chef's Kitchen

Chapter 5 Loops and Files

Topics

1. [5.1 The Increment and Decrement Operators](#)
2. [5.2 The `while` Loop](#)
3. [5.3 Using the `while` Loop for Input Validation](#)
4. [5.4 The `do-while` Loop](#)
5. [5.5 The `for` Loop](#)
6. [5.6 Running Totals and Sentinel Values](#)
7. [5.7 Nested Loops](#)
8. [5.8 The `break` and `continue` Statements](#)
9. [5.9 Deciding Which Loop to Use](#)
10. [5.10 Introduction to File Input and Output](#)
11. [5.11 Common Errors to Avoid](#)

5.1 The Increment and Decrement Operators

Concept:

`++` and `--` are operators that add and subtract one from their operands.



VideoNote The while Loop

To increment a value means to increase it by one, and to decrement a value means to decrease it by one. Both of the following statements increment the variable `number`:

```
number = number + 1;  
number += 1;
```

And `number` is decremented in both of the following statements:

```
number = number - 1;  
number -= 1;
```

Java provides a set of simple unary operators designed just for incrementing and decrementing variables. The increment operator is `++` and the decrement operator is `--`. The following statement uses the `++` operator to increment `number`.

```
number++;
```

And the following statement decrements `number`.

```
number--;
```



Note:

The expression `number++` is pronounced “number plus plus,” and `number--` is pronounced “number minus minus.”

The program in [Code Listing 5-1](#) demonstrates the `++` and `--` operators.

Code Listing 5-1 (`IncrementDecrement.java`)

```
1 /**
2  * This program demonstrates the ++ and -- operators.
3 */
4
5 public class IncrementDecrement
6 {
7     public static void main(String[] args)
8     {
9         int number = 4;
10
11     // Display the value in number.
12     System.out.println("number is " + number);
13     System.out.println("I will increment number.");
14
15     // Increment number.
16     number++;
17
18     // Display the value in number.
19     System.out.println("Now, number is " + number);
20     System.out.println("I will decrement number.");
21
22     // Decrement number.
23     number--;
24
25     // Display the value in number.
26     System.out.println("Now, number is " + number);
27 }
28 }
```

Program Output

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

The statements in [Code Listing 5-1](#) show the increment and decrement operators used in *postfix mode*, which means the operator is placed after the variable. The operators also work in *prefix mode*, where the operator is placed before the variable name:

```
++number;
--number;
```

In both postfix and prefix modes, these operators add one to or subtract one from their operand. [Code Listing 5-2](#) demonstrates this method.

Code Listing 5-2 (Prefix.java)

```
1 /**
2  * This program demonstrates the ++ and -- operators
3  * in prefix mode.
4 */
5
6 public class Prefix
7 {
8     public static void main(String[] args)
9     {
10         int number = 4;
11
12         // Display the value in number.
13         System.out.println("number is " + number);
14         System.out.println("I will increment number.");
15
16         // Increment number.
17         ++number;
18
19         // Display the value in number.
20         System.out.println("Now, number is " + number);
21         System.out.println("I will decrement number.");
```

```
22
23     // Decrement number.
24     --number;
25
26     // Display the value in number.
27     System.out.println("Now, number is " + number);
28 }
29 }
```

Program Output

```
number is 4
I will increment number.
Now, number is 5
I will decrement number.
Now, number is 4
```

The Difference between Postfix and Prefix Modes

In [Code Listings 5-1](#) and [5-2](#), the statements `number++` and `++number` both increment the variable `number`, while the statements `number--` and `--number` both decrement the variable `number`. In these simple statements, it doesn't matter whether the operator is used in postfix or prefix mode. The difference is important, however, when these operators are used in statements that do more than just incrementing or decrementing. For example, look at the following code:

```
number = 4;
System.out.println(number++);
```

The statement calling the `println` method is doing two things: (1) displaying the value of `number` and (2) incrementing `number`. But which happens first? The `println` method will display a different value if `number` is incremented first, than if `number` is incremented last. The answer depends upon the mode of the increment operator.

Postfix mode causes the increment to happen after the value of the variable is

used in the expression. In the previously shown statement, the `println` method will display 4, then `number` will be incremented to 5. Prefix mode, however, causes the increment to happen first. Here is an example:

```
number = 4;  
System.out.println(++number);
```

In these statements, `number` is incremented to 5, then `println` will display 5. For another example, look at the following code:

```
int x = 1, y;  
y = x++; // Postfix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. The second statement does two things:

- It assigns the value of `x` to the variable `y`.
- The variable `x` is incremented.

The value that will be stored in `y` depends on when the increment takes place. Because the `++` operator is used in postfix mode, it acts after the assignment takes place. So, this code will store 1 in `y`. After the code has executed, `x` will contain 2. Let's look at the same code, but with the `++` operator used in prefix mode:

```
int x = 1, y;  
y = ++x; // Prefix increment
```

The first statement declares the variable `x` (initialized with the value 1) and the variable `y`. The second statement does two things:

- The variable `x` is incremented.
- The value of `x` is assigned to the variable `y`.

Because the operator is used in prefix mode, it acts on the variable before the assignment takes place. So, this code will store 2 in `y`. After the code has executed, `x` will also contain 2.



Checkpoint

1. 5.1 What will the following program segments display?

```
1. x = 2;  
   y = x++;  
   System.out.println(y);  
  
2. x = 2;  
   System.out.println(x++);  
  
3. x = 2;  
   System.out.println(--x);  
  
4. x = 8;  
   y = x--;  
   System.out.println(y);
```

5.2 The `while` Loop

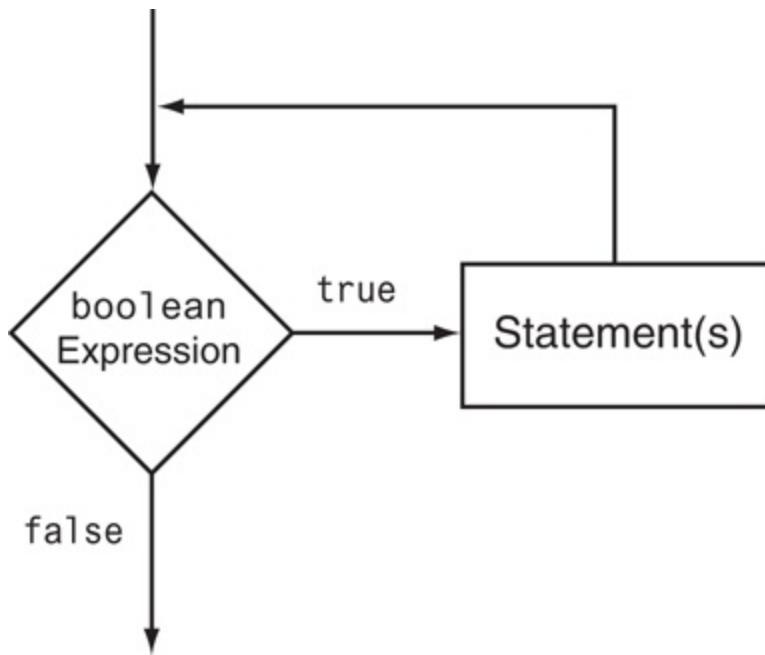
Concept:

A loop is a part of a program that repeats.

In [Chapter 4](#), you were introduced to the concept of control structures, which direct the flow of a program. A *loop* is a control structure that causes a statement or group of statements to repeat. Java has three looping control structures: the `while` loop, the `do-while` loop, and the `for` loop. The difference between each of these is how they control the repetition. In this section, we will focus on the `while` loop.

The `while` loop has two important parts: (1) a boolean expression that is tested for a `true` or `false` value, and (2) a statement or block of statements that is repeated as long as the expression is `true`. [Figure 5-1](#) shows the logic of a `while` loop.

Figure 5-1 Logic of a `while` loop



[Figure 5-1 Full Alternative Text](#)

Here is the general format of the `while` loop:

```
while (BooleanExpression)
    statement;
```

The first line shown in the format is sometimes called the *loop header*. It consists of the key word `while` followed by a boolean expression enclosed in parentheses. The *BooleanExpression* is tested, and if it is *true*, the *statement* is executed. Then, the *BooleanExpression* is tested again. If it is *true*, the *statement* is executed again. This cycle repeats until the boolean expression is *false*. The statement that is repeated is known as the *body* of the loop. It is also considered a conditionally executed statement because it is only executed under the condition that the boolean expression is *true*.

Notice there is no semicolon at the end of the loop header. Like the `if` statement, the `while` loop is not complete without the conditionally executed statement that follows it.

If you wish for the `while` loop to repeat a block of statements, the format is:

```
while (BooleanExpression)
{
```

```
    statement;
    statement;
    // Place as many statements here
    // as necessary.
}
```

The `while` loop works like an `if` statement that executes over and over. As long as the expression in the parentheses is `true`, the conditionally executed statement or block will repeat. The program in [Code Listing 5-3](#) uses the `while` loop to print “Hello” five times.

Code Listing 5-3 (WhileLoop.java)

```
1 /**
2  * This program demonstrates the while loop.
3 */
4
5 public class WhileLoop
6 {
7     public static void main(String[] args)
8     {
9         int number = 1;
10
11     while (number <= 5)
12     {
13         System.out.println("Hello");
14         number++;
15     }
16
17     System.out.println("That's all!");
18 }
19 }
```

Program Output

```
Hello
Hello
Hello
Hello
Hello
That's all!
```

Let's take a closer look at this program. In line 9, an integer variable, `number`, is declared and initialized with the value 1. In line 11, the `while` loop begins with this statement:

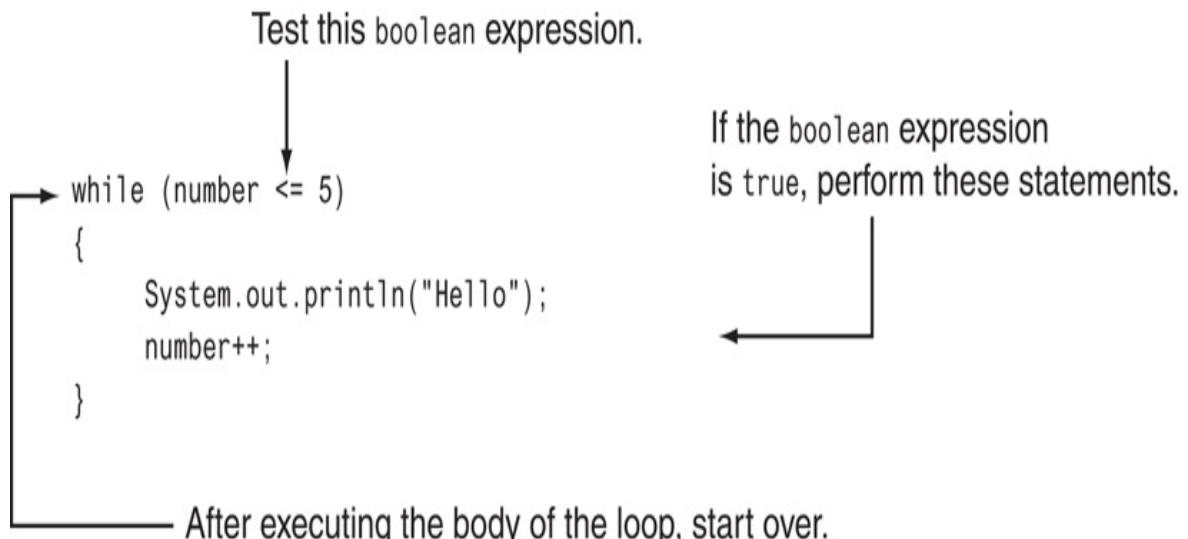
```
while (number <= 5)
```

This statement tests the variable `number` to determine whether it is less than or equal to 5. If it is, then the statements in the body of the loop, which are in lines 13 and 14, are executed:

```
System.out.println("Hello");  
number++;
```

The first statement in the body of the loop (line 13) prints the word “Hello”. The second statement (line 14) uses the increment operator to add one to `number`. This is the last statement in the body of the loop, so after it executes, the loop starts over. It tests the boolean expression again, and if it is `true`, the statements in the body of the loop are executed. This cycle repeats until the boolean expression `number <= 5` is `false`. This is illustrated in [Figure 5-2](#).

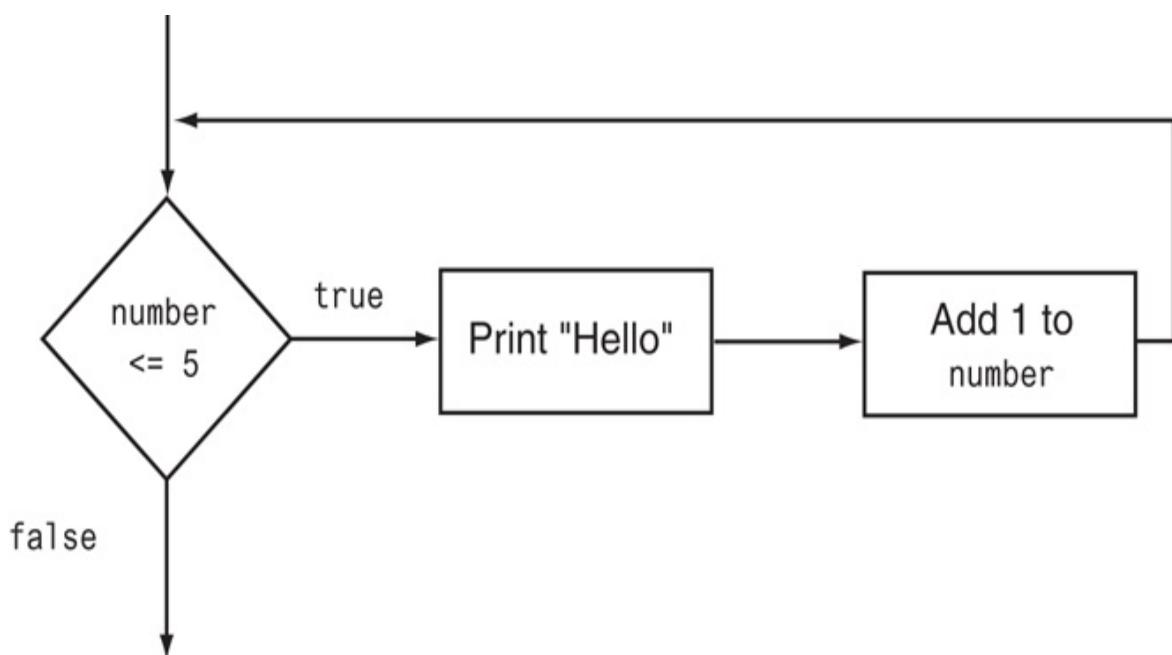
Figure 5-2 The `while` Loop



[Figure 5-2 Full Alternative Text](#)

Each repetition of a loop is known as an *iteration*. This loop will perform five iterations because the variable `number` is initialized with the value 1, and it is incremented each time the body of the loop is executed. When the expression `number <= 5` is tested and found to be `false`, the loop will terminate and the program will resume execution at the statement that immediately follows the loop. [Figure 5-3](#) shows the logic of this loop.

Figure 5-3 Logic of the example while loop



[Figure 5-3 Full Alternative Text](#)

In this example, the `number` variable is referred to as the *loop control variable* because it controls the number of times that the loop iterates.

The `while` Loop Is a Pretest Loop

The `while` loop is known as a *pretest* loop, which means it tests its expression

before each iteration. Notice the variable declaration of `number` in line 9 of [Code Listing 5-3](#):

```
int number = 1;
```

The `number` variable is initialized with the value 1. If `number` had been initialized with a value that is greater than 5, as shown in the following program segment, the loop would never execute:

```
int number = 6;
while (number <= 5)
{
    System.out.println("Hello");
    number++;
}
```

An important characteristic of the `while` loop is that the loop will never iterate if the boolean expression is `false` to begin with. If you want to be sure that a `while` loop executes the first time, you must initialize the relevant data in such a way that the boolean expression starts out as `true`.

Infinite Loops

In all but rare cases, loops must contain within themselves a way to terminate. This means that something inside the loop must eventually make the boolean expression `false`. The loop in [Code Listing 5-3](#) stops when the variable `number` is no longer less than or equal to 5.

If a loop does not have a way of stopping, it is called an infinite loop. An infinite loop continues to repeat until the program is interrupted. Here is an example of an infinite loop:

```
int number = 1;
while (number <= 5)
{
    System.out.println("Hello");
}
```

This is an infinite loop, because it does not contain a statement that changes

the value of the `number` variable. Each time the boolean expression is tested, `number` will contain the value 1.

It's also possible to create an infinite loop by accidentally placing a semicolon after the first line of the `while` loop. Here is an example:

```
int number = 1;
while (number <= 5); // This semicolon is an ERROR!
{
    System.out.println("Hello");
    number++;
}
```

The semicolon at the end of the first line is assumed to be a `null` statement and disconnects the `while` statement from the block that comes after it. To the compiler, this loop looks like:

```
while (number <= 5);
```

This `while` loop will forever execute the `null` statement, which does nothing. The program will appear to have “gone into space” because there is nothing to display screen output or show activity.

Don't Forget the Braces with a Block of Statements

If you're using a block of statements, don't forget to enclose all of the statements in a set of braces. If the braces are accidentally left out, the `while` statement conditionally executes only the very next statement. For example, look at the following code:

```
int number = 1;
// This loop is missing its braces!
while (number <= 5)
    System.out.println("Hello");
    number++;
```

In this code, the `number++` statement is not in the body of the loop. Because

the braces are missing, the `while` statement only executes the statement that immediately follows it. This loop will execute infinitely, because there is no code in its body that changes the `number` variable.

Programming Style and the `while` Loop

It's possible to create loops that look like this:

```
while (number != 99) number = keyboard.nextInt();
```

as well as this:

```
while (number <= 5) { System.out.println("Hello"); number++; }
```

Avoid this style of programming. The programming style you should use with the `while` loop is similar to that of the `if` statement:

- If there is only one statement repeated by the loop, it should appear on the line after the `while` statement and be indented one additional level. The statement can optionally appear inside a set of braces.
- If the loop repeats a block, each line inside the braces should be indented.

This programming style should visually set the body of the loop apart from the surrounding code. In general, you'll find a similar style being used with the other types of loops presented in this chapter.

In the Spotlight: Designing a program with a `while` Loop



A project currently underway at Chemical Labs, Inc. requires that a substance be continually heated in a vat. A technician must check the substance's temperature every 15 minutes. If the substance's temperature does not exceed 102.5 degrees Celsius, then the technician does nothing. However, if the temperature is greater than 102.5 degrees Celsius, the technician must turn down the vat's thermostat, wait 5 minutes, and check the temperature again. The technician repeats these steps until the temperature does not exceed 102.5 degrees Celsius. The director of engineering has asked you to write a program that guides the technician through this process.

Here is the algorithm:

1. Prompt the user to enter the substance's temperature.
2. Repeat the following steps as long as the temperature is greater than 102.5 degrees Celsius:
 1. Tell the technician to turn down the thermostat, wait 5 minutes, and check the temperature again.
 2. Prompt the user to enter the substance's temperature.
3. After the loop finishes, tell the technician that the temperature is acceptable and to check it again in 15 minutes.

After reviewing this algorithm, you realize that steps 2(a) and 2(b) should not be performed if the test condition (temperature is greater than 102.5) is false to begin with. The `while` loop will work well in this situation because it will not execute even once if its condition is false. [Code Listing 5-4](#) shows the program.

Code Listing 5-4 (CheckTemperature.java)

```
1 import java.util.Scanner;  
2  
3 /**
```

```

4 * This program assists a technician in the process
5 * of checking a substance's temperature.
6 */
7
8 public class CheckTemperature
9 {
10    public static void main(String[] args)
11    {
12        final double MAX_TEMP = 102.5; // Maximum temperature
13        double temperature;           // To hold the temperature
14
15        // Create a Scanner object for keyboard input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the current temperature.
19        System.out.print("Enter the substance's Celsius temperature");
20        temperature = keyboard.nextDouble();
21
22        // As long as necessary, instruct the technician
23        // to adjust the temperature.
24        while (temperature > MAX_TEMP)
25        {
26            System.out.println("The temperature is too high. Turn the");
27            System.out.println("thermostat down and wait 5 minutes.");
28            System.out.println("Then, take the Celsius temperature ag");
29            System.out.print("and enter it here: ");
30            temperature = keyboard.nextDouble();
31        }
32
33        // Remind the technician to check the temperature
34        // again in 15 minutes.
35        System.out.println("The temperature is acceptable.");
36        System.out.println("Check it again in 15 minutes.");
37    }
38 }

```

Program Output with Example Input Shown in Bold

Enter the substance's Celsius temperature: **104.7** 
 The temperature is too high. Turn the
 thermostat down and wait 5 minutes.
 Then take the Celsius temperature again
 and enter it here: **103.2** 
 The temperature is too high. Turn the
 thermostat down and wait 5 minutes.
 Then take the Celsius temperature again

and enter it here: **102.1** 
The temperature is acceptable.
Check it again in 15 minutes.



Checkpoint

1. 5.2 How many times will "Hello World" be printed in the following program segment?

```
int count = 10;
while (count < 1)
{
    System.out.println("Hello World");
    count++;
}
```

2. 5.3 How many times will "I love Java programming!" be printed in the following program segment?

```
int count = 0;
while (count < 10)
    System.out.println("Hello World");
    System.out.println("I love Java programming!");
```

5.3 Using the `while` Loop for Input Validation

Concept:

The `while` loop can be used to create input routines that repeat until acceptable data is entered.

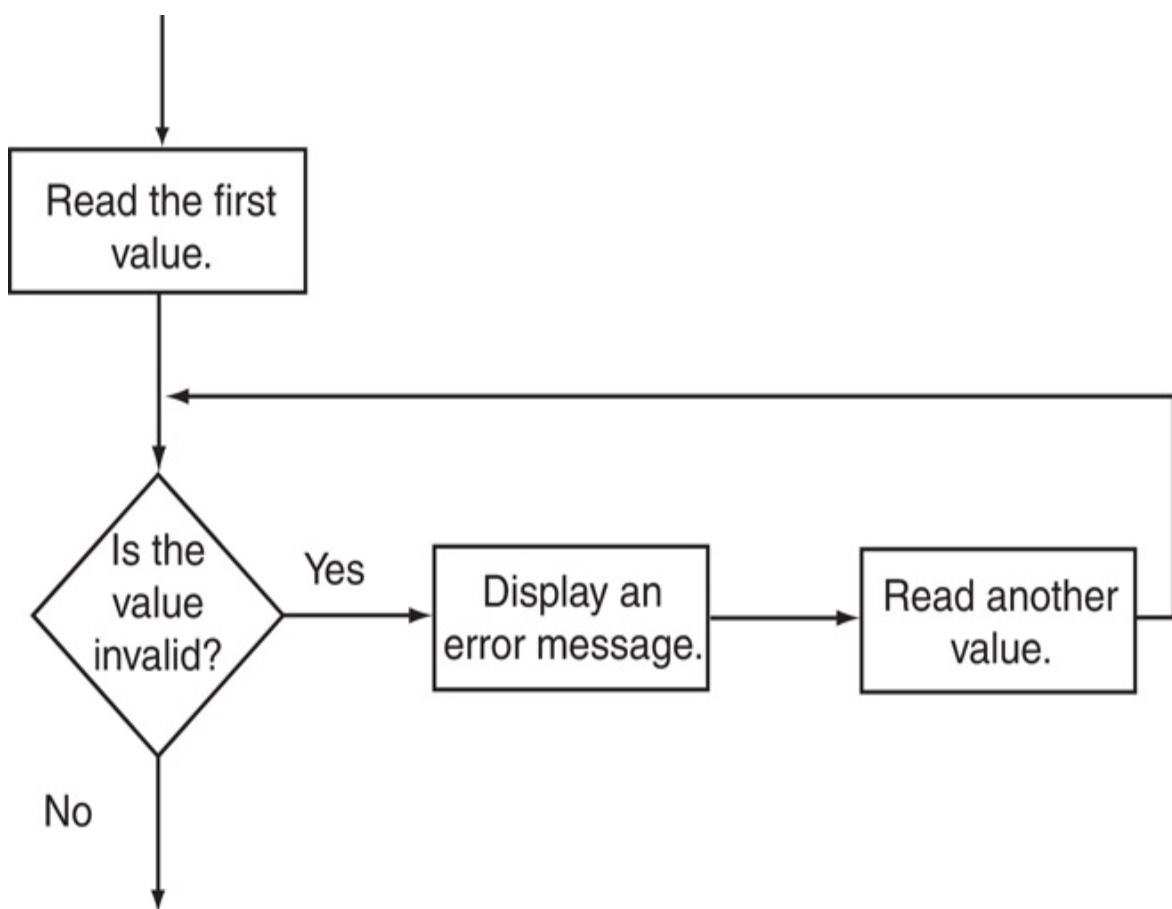
Perhaps the most famous saying of the computer industry is “garbage in, garbage out.” The integrity of a program’s output is only as good as its input, so you should try to make sure garbage does not go into your programs. *Input validation* is the process of inspecting data given to a program by the user and determining if it is valid. A good program should give clear instructions about the kind of input that is acceptable, and should not assume the user has followed those instructions.

The `while` loop is especially useful for validating input. If an invalid value is entered, a loop can require that the user reenter it as many times as necessary. For example, look at the following code, which asks the user to enter a number in the range of 1 through 100:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number in the " +
    "range of 1 through 100: ");
number = keyboard.nextInt();
// Validate the input.
while (number < 1 || number > 100)
{
    System.out.println("That number is invalid.");
    System.out.print("Enter a number in the " +
        "range of 1 through 100: ");
    number = keyboard.nextInt();
}
```

This code first allows the user to enter a number. This takes place just before the loop. If the input is valid, the loop will not execute. If the input is invalid, however, the loop will display an error message, and require the user to enter another number. The loop will continue to execute until the user enters a valid number. The general logic of performing input validation is shown in [Figure 5-4](#).

Figure 5-4 Input validation logic



[Figure 5-4 Full Alternative Text](#)

The read operation that takes place just before the loop is called a *priming read*. It provides the first value for the loop to test. Subsequent values are

obtained by the loop.

The program in [Code Listing 5-5](#) calculates the number of soccer teams a youth league may create, based on a given number of players and a maximum number of players per team. The program uses while loops to validate all of the user input.

Code Listing 5-5 (SoccerTeams.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program calculates the number of soccer teams
5  * that a youth league may create from the number of
6  * available players. Input validation is demonstrated
7  * with while loops.
8 */
9
10 public class SoccerTeams
11 {
12     public static void main(String[] args)
13     {
14         final int MIN_PLAYERS = 9, // Minimum players per team
15             MAX_PLAYERS = 15; // Maximum players per team
16         int players,           // Number of available players
17             teamSize,          // Number of players per team
18             teams,            // Number of teams
19             leftOver;         // Number of left over players
20
21         // Create a scanner object for keyboard input.
22         Scanner keyboard = new Scanner(System.in);
23
24         // Get the number of players per team.
25         System.out.print("Enter the number of players " +
26                         "per team: ");
27         teamSize = keyboard.nextInt();
28
29         // Validate the input.
30         while (teamSize < MIN_PLAYERS || teamSize > MAX_PLAYERS)
31         {
32             System.out.println("You should have at least " +
```

```

33             MIN_PLAYERS
34             " but no more than "
35             MAX_PLAYERS + " per team.");
36     System.out.print("Enter the number of players " +
37             "per team: ");
38     teamSize = keyboard.nextInt();
39 }
40
41 // Get the available number of players.
42 System.out.print("Enter the available number of players: ")
43 players = keyboard.nextInt();
44
45 // Validate the input.
46 while (players < 0)
47 {
48     System.out.println("Please do not enter a negative " +
49             "number.");
50     System.out.print("Enter the available number " +
51             "of players: ");
52     players = keyboard.nextInt();
53 }
54
55 // Calculate the number of teams.
56 teams = players / teamSize;
57
58 // Calculate the number of left over players.
59 leftOver = players % teamSize;
60
61 // Display the results.
62 System.out.println("There will be " + teams + " teams " +
63             "with " + leftOver
64             " players left over.");
65 }
66 }

```

Program Output with Example Input Shown in Bold

Enter the number of players per team: **4** 
 You should have at least 9 but no more than 15 per team.

Enter the number of players per team: **12** 

Enter the available number of players: **-142** 
 Please do not enter a negative number.

Enter the available number of players: **142** 
 There will be 11 teams with 10 players left over.



Checkpoint

1. 5.4 Write an input validation loop that asks the user to enter a number in the range of 10 through 25.
2. 5.5 Write an input validation loop that asks the user to enter ‘Y’, ‘y’, ‘N’, or ‘n’.
3. 5.6 Write an input validation loop that asks the user to enter “Yes” or “No”.

5.4 The do-while Loop

Concept:

The do-while loop is a posttest loop, which means its boolean expression is tested after each iteration.

The do-while loop looks something like an inverted while loop. Here is the do-while loop's format when the body of the loop contains only a single statement:

```
do
    statement;
while (BooleanExpression);
```

Here is the format of the do-while loop when the body of the loop contains multiple statements:

```
do
{
    statement;
    statement;
    // Place as many statements here
    // as necessary.
} while (BooleanExpression);
```



Note:

The do-while loop must be terminated with a semicolon.

The do-while loop is a posttest loop. This means it does not test its boolean expression until it has completed an iteration. As a result, the do-while loop

always performs at least one iteration, even if the boolean expression is false to begin with. This differs from the behavior of a while loop, which you will recall is a pretest loop. For example, in the following while loop, the `println` statement will not execute at all:

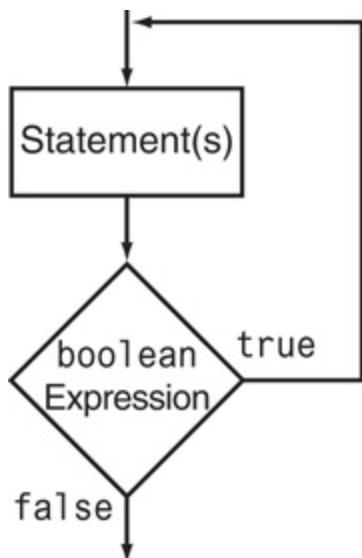
```
int x = 1;  
while (x < 0)  
    System.out.println(x);
```

But the `println` statement in the following do-while loop will execute once, because the do-while loop does not evaluate the expression `x < 0` until the end of the iteration.

```
int x = 1;  
do  
    System.out.println(x);  
while (x < 0);
```

[Figure 5-5](#) illustrates the logic of the do-while loop.

Figure 5-5 Logic of the do-while loop



[Figure 5-5 Full Alternative Text](#)

You should use the do-while loop when you want to make sure the loop executes at least once. For example, the program in [Code Listing 5-6](#) averages a series of three test scores for a student. After the average is displayed, it asks the user if he or she wants to average another set of test scores. The program repeats as long as the user enters Y for yes.

Code Listing 5-6 (TestAverage1.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates a user-controlled loop.
5  */
6
7 public class TestAverage1
8 {
9     public static void main(String[] args)
10    {
11        String input;          // To hold keyboard input
12        double score1, score2, score3; // Three test scores
13        double average;         // Average test score
14        char repeat;           // Holds 'y' or 'n'
15
16        // Create a Scanner object for keyboard input.
17        Scanner keyboard = new Scanner(System.in);
18
19        System.out.println("This program calculates the average " +
20                           "of three test scores.");
21
22        do
23        {
24            // Get the three test scores.
25            System.out.print("Enter score #1: ");
26            score1 = keyboard.nextDouble();
27            System.out.print("Enter score #2: ");
28            score2 = keyboard.nextDouble();
29            System.out.print("Enter score #3: ");
30            score3 = keyboard.nextDouble();
31
32            // Calculate and print the average test score.
33            average = (score1 + score2 + score3) / 3.0;
```

```
34     System.out.println("The average is " + average);
35     System.out.println();
36
37     // Does the user want to average another set?
38     System.out.println("Would you like to average " +
39                         "another set of test scores?");
40     System.out.print("Enter Y for yes or N for no: ");
41     input = keyboard.next(); // Read a string.
42     repeat = input.charAt(0); // Get the first char.
43
44 } while (repeat == 'Y' || repeat == 'y');
45 }
46 }
```

Program Output with Example Input Shown in Bold

This program calculates the average of three test scores.

Enter score #1: **89** 

Enter score #2: **90** 

Enter score #3: **97** 

The average is 92.0

Would you like to average another set of test scores?

Enter Y for yes or N for no: **Y** 

Enter score #1: **78** 

Enter score #2: **65** 

Enter score #3: **88** 

The average is 77.0

Would you like to average another set of test scores?

Enter Y for yes or N for no: **N** 

When this program was written, the programmer had no way of knowing the number of times the loop would iterate. This is because the loop asks the user if he or she wants to repeat the process. This type of loop is known as a *user-controlled loop* because it allows the user to decide the number of iterations.

5.5 The for Loop

Concept:

The `for` loop is ideal for performing a known number of iterations.

In general, there are two categories of loops: conditional loops and count-controlled loops. A *conditional loop* executes as long as a particular condition exists. For example, an input validation loop executes as long as the input value is invalid. When you write a conditional loop, you have no way of knowing the number of times it will iterate.

Sometimes, you do know the exact number of iterations that a loop must perform. A loop that repeats a specific number of times is known as a *count-controlled loop*. For example, if a loop asks the user to enter the sales amounts for each month in the year, it will iterate 12 times. In essence, the loop counts to 12, and asks the user to enter a sales amount each time it makes a count.

A count-controlled loop must possess three elements:

1. It must initialize a control variable to a starting value.
2. It must test the control variable by comparing it to a maximum value. When the control variable reaches its maximum value, the loop terminates.
3. It must update the control variable during each iteration. This is usually done by incrementing the variable.

In Java, the `for` loop is ideal for writing count-controlled loops. It is specifically designed to initialize, test, and update a loop control variable.

Here is the general format of the `for` loop when used to repeat a single statement:

```
for (Initialization; Test; Update)
    statement;
```

The format of the `for` loop when used to repeat a block is:

```
for (Initialization; Test; Update)
{
    statement;
    statement;
    // Place as many statements here
    // as necessary.
}
```

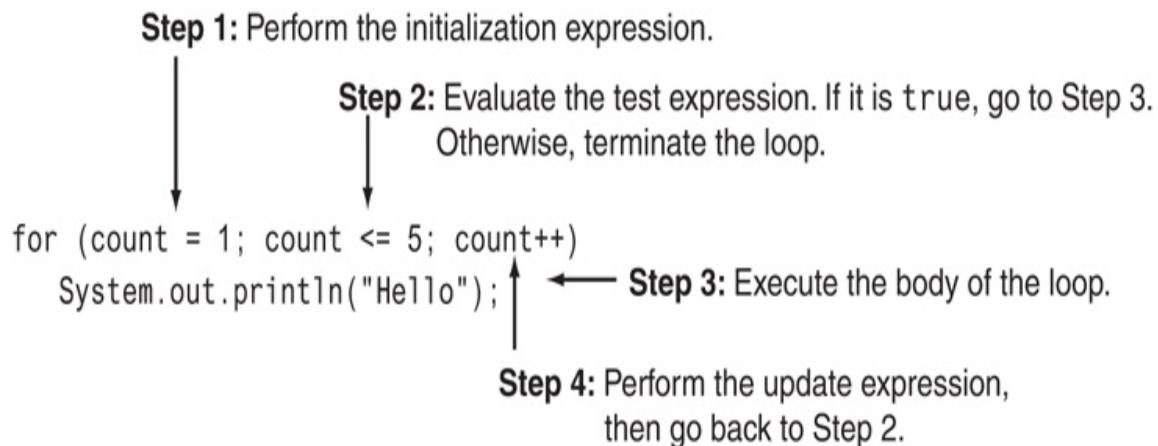
The first line of the `for` loop is known as the *loop header*. After the key word `for`, there are three expressions inside the parentheses, separated by semicolons. (Notice there is no semicolon after the third expression.) The first expression is the *initialization expression*. It is normally used to initialize a control variable to its starting value. This is the first action performed by the loop, and it is only done once. The second expression is the *test expression*. This is a boolean expression that controls the execution of the loop. As long as this expression is `true`, the body of the `for` loop will repeat. The `for` loop is a pretest loop, so it evaluates the test expression before each iteration. The third expression is the *update expression*. It executes at the end of each iteration. Typically, this is a statement that increments the loop's control variable.

Here is an example of a simple `for` loop that prints “Hello” five times:

```
for (count = 1; count <= 5; count++)
    System.out.println("Hello");
```

In this loop, the initialization expression is `count = 1`, the test expression is `count <= 5`, and the update expression is `count++`. The body of the loop has one statement, which is the `println` statement. [Figure 5-6](#) illustrates the sequence of events that take place during the loop's execution. Notice that Steps 2 through 4 are repeated as long as the test expression is `true`.

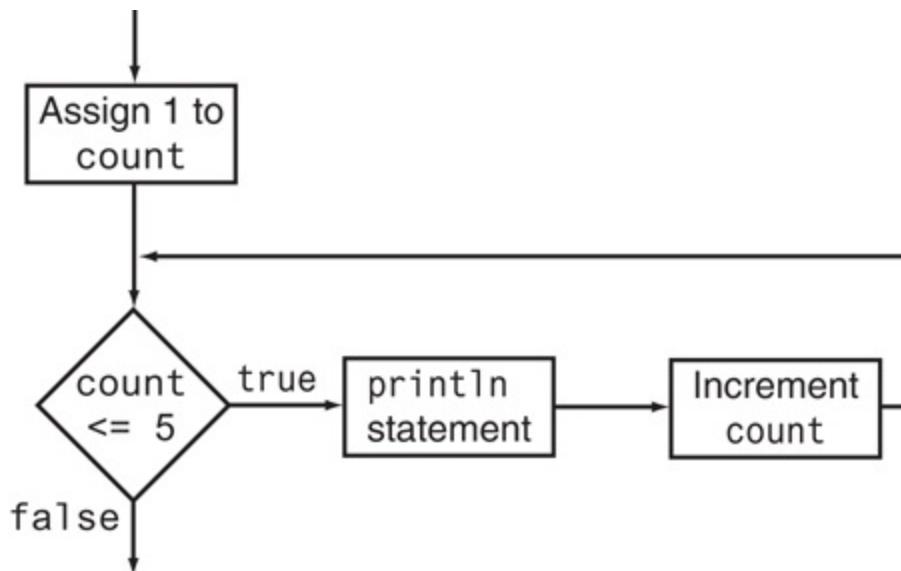
Figure 5-6 Sequence of events in the for loop



[Figure 5-6 Full Alternative Text](#)

[Figure 5-7](#) shows the loop's logic in the form of a flowchart.

Figure 5-7 Logic of the for loop



[Figure 5-7 Full Alternative Text](#)

Notice how the control variable, count, is used to control the number of times that the loop iterates. During the execution of the loop, this variable takes on the values 1 through 5, and when the test expression count ≤ 5 is false, the loop terminates. Because this variable keeps a count of the number of iterations, it is often called a *counter variable*.

Also notice that the count variable is used only in the loop header, to control the number of loop iterations. It is not used for any other purpose. It is also possible to use the control variable within the body of the loop. For example, look at the following code:

```
for (number = 1; number <= 10; number++)
    System.out.print(number + " ");
```

The control variable in this loop is number. In addition to controlling the number of iterations, it is also used in the body of the loop. This loop will produce the following output:

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

As you can see, the loop displays the contents of the number variable during each iteration. The program in [Code Listing 5-7](#) shows another example of a for loop that uses its control variable within the body of the loop. This program displays a table showing the numbers 1 through 10 and their squares.

Code Listing 5-7 (Squares.java)

```
1 /**
2  * This program demonstrates the for loop.
3 */
4
5 public class Squares
6 {
7     public static void main(String[] args)
8     {
9         int number; // Loop control variable
```

```
10
11     System.out.println("Number    Number Squared");
12     System.out.println("-----");
13
14     for (number = 1; number <= 10; number++)
15     {
16         System.out.println(number + "\t\t" +
17                             number * number);
18     }
19 }
20 }
```

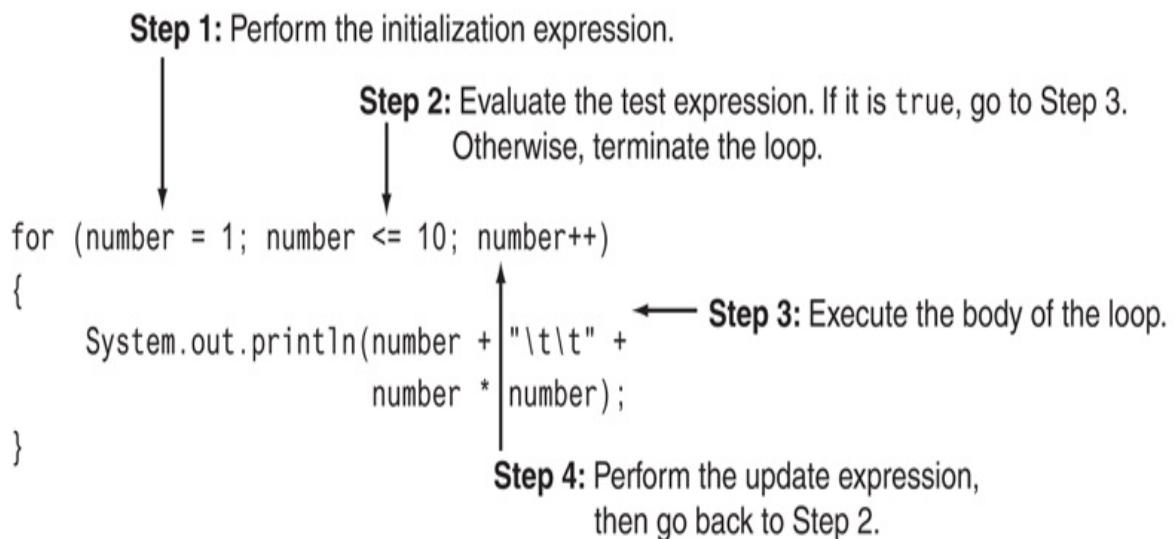
Program Output

Number Number Squared

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

[Figure 5-8](#) illustrates the sequence of events performed by this `for` loop.

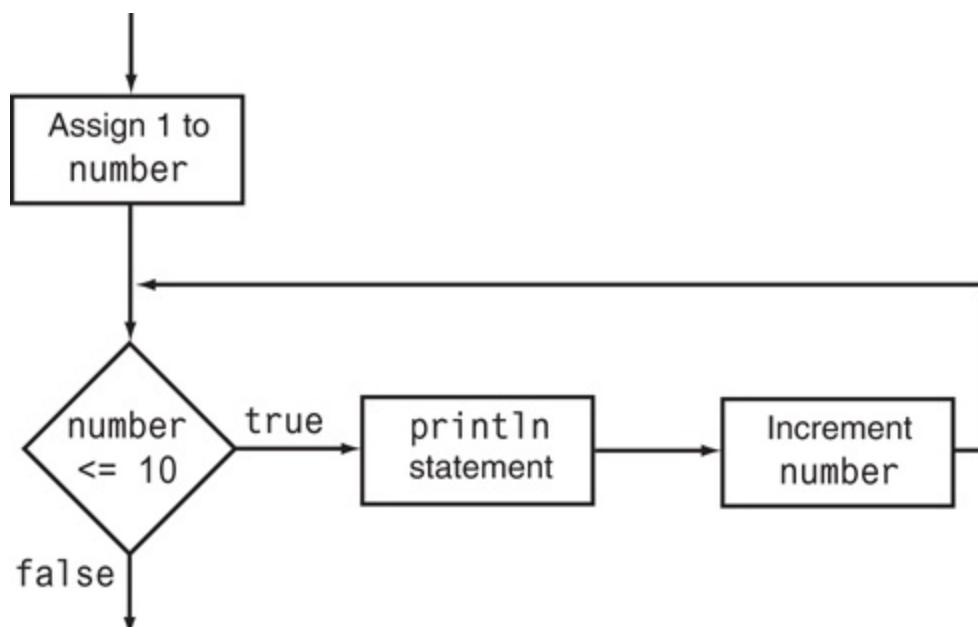
**Figure 5-8 Sequence of events
with the `for` loop in [Code](#)
[Listing 5-7](#)**



[Figure 5-8 Full Alternative Text](#)

[Figure 5-9](#) shows the logic of the loop.

Figure 5-9 Logic of the for loop in [Code Listing 5-7](#)



[Figure 5-9 Full Alternative Text](#)

The for Loop Is a Pretest Loop

Because the `for` loop tests its boolean expression before it performs an iteration, it is a pretest loop. It is possible to write a `for` loop in such a way that it will never iterate. Here is an example:

```
for (count = 11; count <= 10; count++)
    System.out.println("Hello");
```

Because the variable `count` is initialized to a value that makes the boolean expression `false` from the beginning, this loop terminates as soon as it begins.

Avoid Modifying the Control Variable in the Body of the for Loop

Be careful not to place a statement that modifies the control variable in the body of the `for` loop. All modifications of the control variable should take place in the update expression, which is automatically executed at the end of each iteration. If a statement in the body of the loop also modifies the control variable, the loop will probably not terminate when you expect it to. The following loop, for example, increments `x` twice for each iteration:

```
for (x = 1; x <= 10; x++)
{
    System.out.println(x);
    x++;
}
```

Other Forms of the Update Expression

You are not limited to using increment statements in the update expression. Here is a loop that displays all the even numbers from 2 through 100 by adding 2 to its counter:

```
for (number = 2; number <= 100; number += 2)
    System.out.println(number);
```

And here is a loop that counts backward from 10 down to 0:

```
for (number = 10; number >= 0; number--)
    System.out.println(number);
```

Declaring a Variable in the for Loop's Initialization Expression

Not only may the control variable be initialized in the initialization expression, it may be declared there as well. The following code shows an example. This is a modified version of the loop in [Code Listing 5-7](#).

```
for (int number = 1; number <= 10; number++)
{
    System.out.println(number + "\t\t" +
        number * number);
}
```

In this loop, the variable `number` is both declared and initialized in the initialization expression. If the control variable is used only in the loop, it makes sense to declare it in the loop header. This makes the variable's purpose more clear.

When a variable is declared in the initialization expression of a `for` loop, the scope of the variable is limited to the loop. This means you cannot access the variable in statements outside the loop. For example, the following program segment will not compile because the last `println` statement cannot access the variable `count`.

```
for (int count = 1; count <= 10; count++)
    System.out.println(count);
```

```
System.out.println("count is now " + count); // ERROR!
```

Creating a User Controlled for Loop

Sometimes, you want the user to determine the maximum value of the control variable in a `for` loop, and therefore determine the number of times the loop iterates. For example, look at the program in [Code Listing 5-8](#). It is a modification of [Code Listing 5-7](#). Instead of displaying the numbers 1 through 10 and their squares, this program allows the user to enter the maximum value to display.

Code Listing 5-8 (UserSquares.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates a user-controlled
5  * for loop.
6 */
7
8 public class UserSquares
9 {
10    public static void main(String[] args)
11    {
12        int number, // Loop control variable
13        maxValue; // Maximum value to display
14
15        // Create a Scanner object for keyboard input.
16        Scanner keyboard = new Scanner(System.in);
17
18        System.out.println("I will display a table of " +
19                           "numbers and their squares.");
20
21        // Get the maximum value to display.
22        System.out.print("How high should I go? ");
23        maxValue = keyboard.nextInt();
```

```
24  
25     // Display the table.  
26     System.out.println("Number  Number Squared");  
27     System.out.println("-----");  
28  
29     for (number = 1; number <= maxValue; number++)  
30     {  
31         System.out.println(number + "\t\t" +  
32                         number * number);  
33     }  
34 }  
35 }
```

Program Output with Example Input Shown in Bold

I will display a table of numbers and their squares.

How high should I go? **7** 

Number Number Squared

```
-----  
1      1  
2      4  
3      9  
4      16  
5      25  
6      36  
7      49
```

In lines 22 and 23, which are before the loop, this program asks the user to enter the highest value to display. This value is stored in the `maxValue` variable.

```
System.out.print("How high should I go? ");  
maxValue = keyboard.nextInt();
```

In line 29, the `for` loop's test expression uses this value as the upper limit for the control variable:

```
for (number = 1; number <= maxValue; number++)
```

In this loop, the `number` variable takes on the values 1 through `maxValue`, then the loop terminates.

Using Multiple Statements in the Initialization and Update Expressions

It is possible to execute more than one statement in the initialization expression and the update expression. When using multiple statements in either of these expressions, simply separate the statements with commas. For example, look at the loop in the following code, which has two statements in the initialization expression.

```
int x, y;
for (x = 1, y = 1; x <= 5; x++)
{
    System.out.println(x + " plus " + y +
        " equals " +
        (x + y));
}
```

This loop's initialization expression is

```
x = 1, y = 1
```

This initializes two variables, x and y. The output produced by this loop is

```
1 plus 1 equals 2
2 plus 1 equals 3
3 plus 1 equals 4
4 plus 1 equals 5
5 plus 1 equals 6
```

We can further modify the loop to execute two statements in the update expression. Here is an example:

```
int x, y;
for (x = 1, y = 1; x <= 5; x++, y++)
{
    System.out.println(x + " plus " + y +
        " equals " +
        (x + y));
```

}

The loop's update expression is

`x++, y++`

This update expression increments both the `x` and `y` variables. The output produced by this loop is

```
1 plus 1 equals 2
2 plus 2 equals 4
3 plus 3 equals 6
4 plus 4 equals 8
5 plus 5 equals 10
```

Connecting multiple statements with commas works well in the initialization and update expressions, but don't try to connect multiple boolean expressions this way in the test expression. If you wish to combine multiple boolean expressions in the test expression, you must use the `&&` or `||` operators.

In the Spotlight: Designing a Count-Controlled for Loop



Your friend Amanda just inherited a European sports car from her uncle. Amanda lives in the United States, and she is afraid she will get a speeding ticket because the car's speedometer indicates kilometers per hour. She has asked you to write a program that displays a table of speeds in kilometers per hour with their values converted to miles per hour. The formula for converting kilometers per hour (KPH) to miles per hour (MPH) is

$$\text{MPH} = \text{KPH} * 0.6214$$

The table that your program displays should show speeds from 60 kilometers

per hour through 130 kilometers per hour, in increments of 10, along with their values converted to miles per hour. The table should look something like this:

KPH MPH

60 37.3

70 43.5

80 49.7

.

.

.

130 80.8

After thinking about this table of values, you decide that you will write the following code:

- A class named SpeedConverter that can perform the conversion from kilometers per hour to miles per hour.
- A program that creates an instance of the SpeedConverter class and uses that object to convert the sequence of kilometer per hour speeds to miles per hour.

The SpeedConverter class, which is shown in [Code Listing 5-9](#), has only one method named getMPH. The method, which appears in lines 18 through 21, accepts a speed in kilometers per hour as an argument, and it returns that speed converted to miles per hour.

Code Listing 5-9 (SpeedConverter.java)

```
1 /**
2  * The SpeedConverter class converts speeds
3  * in KPH (kilometers per hour) to MPH (miles
4  * per hour).
```

```

5  */
6
7 public class SpeedConverter
8 {
9     // Factor to convert MPH to KPH
10    final double CONVERSION_FACTOR = 0.6214;
11
12    /**
13     * The getMPH method accepts a speed in
14     * KPH as an argument and returns that
15     * speed converted to MPH.
16    */
17
18    public double getMPH(double kph)
19    {
20        return kph * CONVERSION_FACTOR;
21    }
22 }
```

As previously mentioned, the program that displays the table of speeds creates an instance of the SpeedConverter class. In the program, you decide to write a `for` loop that uses a counter variable to hold the kilometer-per-hour speeds. The counter's starting value is 60 and its ending value is 130, and you need to add 10 to the counter variable after each iteration. Inside the loop, you will call the SpeedConverter class's `getMPH` method, passing the counter variable as an argument. The method will return the speed converted to miles per hour. [Code Listing 5-10](#) shows the code.

Code Listing 5-10 (SpeedTable.java)

```

1 /**
2  * This program displays a table of speeds in
3  * kph converted to mph.
4 */
5
6 public class SpeedTable
7 {
8     public static void main(String[] args)
9     {
10        // Constants
```

```

11     final int STARTING_KPH = 60; // Starting speed
12     final int MAX_KPH = 130;    // Maximum speed
13     final int INCREMENT = 10;   // Speed increment
14
15     // Variables
16     int kph;      // To hold the speed in kph
17     double mph;   // To hold the speed in mph
18
19     // Create an instance of the SpeedConverter class.
20     SpeedConverter converter = new SpeedConverter();
21
22     // Display the table headings.
23     System.out.println("KPH\t\tMPH");
24     System.out.println("-----");
25
26     // Display the speeds.
27     for (kph = STARTING_KPH; kph <= MAX_KPH; kph += INCREMENT)
28     {
29         // Get the mph.
30         mph = converter.getMPH(kph);
31
32         // Display the speeds in kph and mph.
33         System.out.printf("%d\t\t%.1f\n", kph, mph);
34     }
35 }
36 }
```

Program Output

KPH	MPH

60	37.3
70	43.5
80	49.7
90	55.9
100	62.1
110	68.4
120	74.6
130	80.8



Checkpoint

1. 5.7 Name the three expressions that appear inside the parentheses in the

for loop's header.

2. 5.8 You want to write a for loop that displays “I love to program” 50 times. Assume that you will use a control variable named count.
 1. What initialization expression will you use?
 2. What test expression will you use?
 3. What update expression will you use?
 4. Write the loop.
3. 5.9 What will the following program segments display?
 1.

```
for (int count = 0; count < 6; count++)
    System.out.println(count + count);
```
 2.

```
or (int value = -5; value < 5; value++)
    System.out.println(value);
```
 3.

```
for (int x = 5; x <= 14; x += 3)
    System.out.println(x);
    System.out.println(x);
```
4. 5.10 Write a for loop that displays your name 10 times.
5. 5.11 Write a for loop that displays all of the odd numbers, 1 through 49.
6. 5.12 Write a for loop that displays every fifth number, 0 through 100.

5.6 Running Totals and Sentinel Values

Concept:

A running total is a sum of numbers that accumulates with each iteration of a loop. The variable used to keep the running total is called an accumulator. A sentinel is a value that signals when the end of a list of values has been reached.

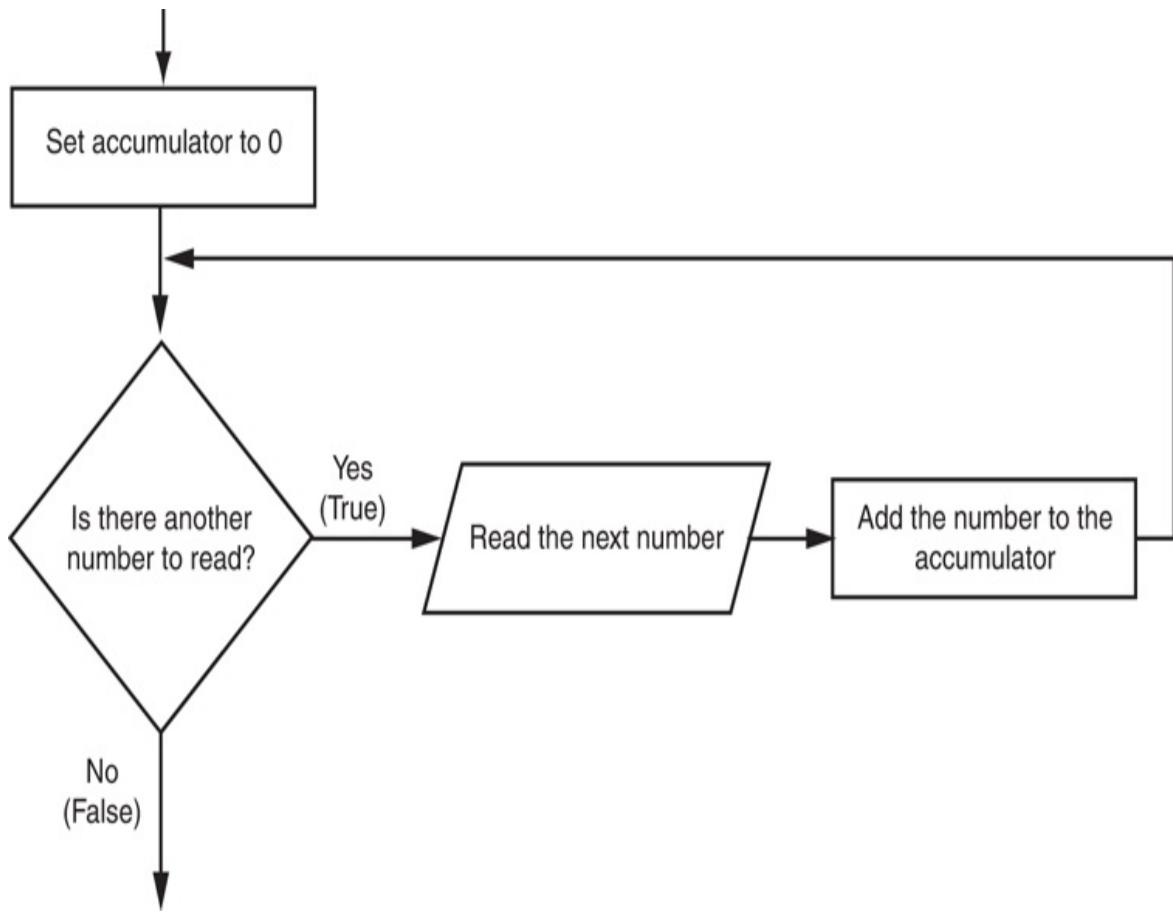
Many programming tasks require you to calculate the total of a series of numbers. For example, suppose you are writing a program that calculates a business's total sales for a week. The program reads the sales for each day as input and calculates the total of those numbers.

Programs that calculate the total of a series of numbers typically use two elements:

- A loop that reads each number in the series.
- A variable that accumulates the total of the numbers as they are read.

The variable that is used to accumulate the total of the numbers is called an *accumulator*. It is often said that the loop keeps a *running total* because it accumulates the total as it reads each number in the series. [Figure 5-10](#) shows the general logic of a loop that calculates a running total.

Figure 5-10 Logic for calculating a running total



[Figure 5-10 Full Alternative Text](#)

When the loop finishes, the accumulator will contain the total of the numbers that were read by the loop. Notice that the first step in the flowchart is to set the accumulator variable to 0. This is a critical step. Each time the loop reads a number, it adds it to the accumulator. If the accumulator starts with any value other than 0, it will not contain the correct total when the loop finishes.

Let's look at a program that calculates a running total. [Code Listing 5-11](#) calculates a company's total sales over a period of time by taking daily sales amounts as input and calculating a running total of them as they are gathered.

Code Listing 5-11 (`TotalSales.java`)

```

1 import java.util.Scanner;
2
3 /**
4  * This program calculates a running total.
5 */
6
7 public class TotalSales
8 {
9     public static void main(String[] args)
10    {
11        int days;      // The number of days
12        double sales; // A day's sales amount
13        double totalSales; // Accumulator
14
15        // Create a Scanner object for keyboard input.
16        Scanner keyboard = new Scanner(System.in);
17
18        // Get the number of days.
19        System.out.print("For how many days do you have " +
20                         "sales amounts? ");
21        days = keyboard.nextInt();
22
23        // Set the accumulator to 0.
24        totalSales = 0.0;
25
26        // Get the sales amounts and calculate
27        // a running total.
28        for (int count = 1; count <= days; count++)
29        {
30            System.out.print("Enter the sales for day " +
31                            count + ": ");
32            sales = keyboard.nextDouble();
33            totalSales += sales; // Add sales to total.
34        }
35
36        // Display the total sales.
37        System.out.printf("The total sales are $%, .2f\n", totalSale
38    }
39 }

```

Program Output with Example Input Shown in Bold

For how many days do you have sales amounts? **5** 

Enter the sales for day 1: **1500** 

Enter the sales for day 2: **2500** 

```
Enter the sales for day 3: 3500 Enter
Enter the sales for day 4: 4500 Enter
Enter the sales for day 5: 5500 Enter
The total sales are $17,500.00
```

Let's take a closer look at this program. In lines 19 and 20, the user is asked to enter the number of days that he or she has sales amounts for. The number of days is read from the keyboard and assigned to the days variable. Next, in line 24, the totalSales variable is assigned 0.0.

In general programming terms, the totalSales variable is referred to as an *accumulator*. An *accumulator* is a variable initialized with a starting value, which is usually zero, then accumulates a sum of numbers by having the numbers added to it. As you will see, it is critical that the accumulator is set to zero before values are added to it.

Next, the for loop that appears in lines 28 through 34 executes:

```
for (int count = 1; count <= days; count++)
{
    System.out.print("Enter the sales for day " +
                      count + ": ");
    sales = keyboard.nextDouble();
    totalSales += sales; // Add sales to total.
}
```

The user enters the daily sales amounts, which are assigned to the sales variable. The contents of sales are then added to the totalSales variable. Because totalSales was initially assigned 0.0, after the first iteration it will be set to the same value as sales. After each subsequent iteration, totalSales will be increased by the amount in sales. After the loop has finished, totalSales will contain the total of all the daily sales amounts entered. Now it should be clear why we assigned 0.0 to totalSales before the loop executed. If totalSales started at any other value, the total would be incorrect.

Using a Sentinel Value

The program in [Code Listing 5-11](#) requires the user to know in advance the number of days for which he or she has sales amounts. Sometimes the user has a list of input values that is very long and doesn't know the number of items there are. In other cases, the user might be entering values from several lists, and it is impractical to require that every item in every list be counted.

A technique that can be used in these situations is to ask the user to enter a sentinel value at the end of the list. A *sentinel value* is a special value that cannot be mistaken as a member of the list and signals that there are no more values to be entered. When the user enters the sentinel value, the loop terminates.

The program in [Code Listing 5-12](#) shows an example. It calculates the total points earned by a soccer team over a series of games. It allows the user to enter the series of game points, then enter -1 to signal the end of the list.

Code Listing 5-12 (SoccerPoints.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program calculates the total number of points a
5  * soccer team has earned over a series of games. The user
6  * enters a series of point values, then -1 when finished.
7  */
8
9 public class SoccerPoints
10 {
11     public static void main(String[] args)
12     {
13         int points,          // Game points
14         totalPoints = 0; // Accumulator
15
16         // Create a Scanner object for keyboard input.
17         Scanner keyboard = new Scanner(System.in);
18
19         // Display general instructions.
20         System.out.println("Enter the number of points your team");
21         System.out.println("has earned for each game this season.")
```

```

22     System.out.println("Enter -1 when finished.");
23     System.out.println();
24
25     // Get the first number of points.
26     System.out.print("Enter game points or -1 to end: ");
27     points = keyboard.nextInt();
28
29     // Accumulate the points until -1 is entered.
30     while (points != -1)
31     {
32         // Add points to totalPoints.
33         totalPoints += points;
34
35         // Get the next number of points.
36         System.out.print("Enter game points or -1 to end: ");
37         points = keyboard.nextInt();
38     }
39
40     // Display the total number of points.
41     System.out.println("The total points are " +
42                         totalPoints);
43 }
44 }
```

Program Output with Example Input Shown in Bold

Enter the number of points your team
has earned for each game this season.
Enter -1 when finished.

```

Enter game points or -1 to end: 7 
Enter game points or -1 to end: 9 
Enter game points or -1 to end: 4 
Enter game points or -1 to end: 6 
Enter game points or -1 to end: 8 
Enter game points or -1 to end: -1 
The total points are 34
```

The value -1 was chosen for the sentinel because it is not possible for a team to score negative points. Notice that this program performs a priming read to get the first value. This makes it possible for the loop to immediately terminate if the user enters -1 as the first value. Also note that the sentinel value is not included in the running total.



Checkpoint

1. 5.13 Write a `for` loop that repeats seven times, asking the user to enter a number. The loop should also calculate the sum of the numbers entered.
2. 5.14 In the following program segment, which variable is the loop control variable (also known as the counter variable) and which is the accumulator?

```
int a, x = 0, y = 0;
Scanner keyboard = new Scanner(System.in);
while (x < 10)
{
    System.out.print("Enter a number: ");
    a = keyboard.nextInt();
    y += a;
}
System.out.println("The sum is " + y);
```

3. 5.15 Why should you be careful when choosing a sentinel value?

5.7 Nested Loops

Concept:

A loop that is inside another loop is called a nested loop.

Nested loops are necessary when a task performs a repetitive operation, and that task itself must be repeated. A clock is a good example of something that works like a nested loop. The second hand, minute hand, and hour hand all spin around the face of the clock. Each time the hour hand increments, the minute hand has incremented 60 times. Each time the minute hand increments, the second hand has incremented 60 times.

The program in [Code Listing 5-13](#) uses nested loops to simulate a clock.

Code Listing 5-13 (clock.java)

```
1 /**
2  * This program uses nested loops to simulate a clock.
3 */
4
5 public class Clock
6 {
7     public static void main(String[] args)
8     {
9         // Simulate the clock.
10        for (int hours = 1; hours <= 12; hours++)
11        {
12            for (int minutes = 0; minutes <= 59; minutes++)
13            {
14                for (int seconds = 0; seconds <= 59; seconds++)
15                {
16                    System.out.printf("%02d:%02d:%02d\n", hours, minutes,
17                }
```

```
18      }
19    }
20  }
21 }
```

Program Output

```
01:00:00
01:00:01
01:00:02
01:00:03
(The loop continues to count . . .)
```

```
12:59:57
12:59:58
12:59:59
```

The innermost loop will iterate 60 times for each single iteration of the middle loop. The middle loop will iterate 60 times for each single iteration of the outermost loop. When the outermost loop has iterated 12 times, the middle loop will have iterated 720 times, and the innermost loop will have iterated 43,200 times.

The simulated clock example brings up a few points about nested loops:

- An inner loop goes through all of its iterations for each iteration of an outer loop.
- Inner loops complete their iterations before outer loops do.
- To get the total number of iterations of a nested loop, multiply the number of iterations of all the loops.

The program in [Code Listing 5-14](#) is another test-averaging program. It asks the user for the number of students and the number of test scores per student. A nested inner loop asks for all the test scores for one student, iterating once for each test score. The outer loop iterates once for each student.

Code Listing 5-14

(TestAverages2.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates a user-controlled loop.
5  */
6
7 public class TestAverages2
8 {
9     public static void main(String[] args)
10    {
11        int numStudents; // Number of students
12        int numTests; // Number of tests per student
13        double score; // Test score
14        double total; // Accumulator for test scores
15        double average; // Average test score
16
17        // Create a Scanner object for keyboard input.
18        Scanner keyboard = new Scanner(System.in);
19
20        System.out.println("This program averages test scores.");
21
22        // Get the number of students.
23        System.out.print("How many students do you have? ");
24        numStudents = keyboard.nextInt();
25
26        // Get the number of test scores per student.
27        System.out.print("How many test scores per student? ");
28        numTests = keyboard.nextInt();
29
30        // Process all the students.
31        for (int student = 1; student <= numStudents; student++)
32        {
33            // Set the accumulator to zero.
34            total = 0.0;
35
36            // Get the test scores for a student.
37            for (int test = 1; test <= numTests; test++)
38            {
39                System.out.print("Enter score " + test +
40                    " for student " + student + ": ");
41                score = keyboard.nextDouble();
42                total += score; // Add score to total.
43            }
44
```

```
45     // Calculate and display the average.  
46     average = total / numTests;  
47     System.out.println("The average score for student " +  
48             student + " is " + average);  
49     System.out.println();  
50 }  
51 }  
52 }
```

Program Output with Example Input Shown in Bold

This program averages test scores.

How many students do you have? **2**

How many test scores per student? **3**

Enter score 1 for student 1: **78**

Enter score 2 for student 1: **86**

Enter score 3 for student 1: **91**

The average score for student 1 is 85.0

Enter score 1 for student 2: **97**

Enter score 2 for student 2: **88**

Enter score 3 for student 2: **91**

The average score for student 2 is 92.0

In the Spotlight: Using Nested Loops to Print Patterns



One interesting way to learn about nested loops is to use them to display patterns on the screen. Let's look at a simple example. Suppose we want to print asterisks on the screen in the following rectangular pattern:

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

If you think of this pattern as having rows and columns, you can see that it has eight rows, and each row has six columns. The following code can be used to display one row of asterisks:

```
final int COLS = 6;
for (int col = 0; col < COLS; col++)
{
    System.out.print("*");
}
```

If we run this code in a program, it will produce the following output:

```
*****
```

To complete the entire pattern, we need to execute this loop eight times. We can place the loop inside another loop that iterates eight times, as shown here:

```
1 final int COLS = 6;
2 final int ROWS = 8;
3 for (int row = 0; row < ROWS; row++)
4 {
5     for (int col = 0; col < COLS; col++)
6     {
7         System.out.print("*");
8     }
9     System.out.println();
10 }
```

The outer loop iterates eight times. Each time it iterates, the inner loop

iterates six times. (Notice that in line 9, after each row has been printed, we call the `System.out.println()` method. We have to do that to advance the screen cursor to the next line at the end of each row. Without that statement, all the asterisks will be printed in one long row on the screen.)

We could easily write a program that prompts the user for the number of rows and columns, as shown in [Code Listing 5-15](#).

Code Listing 5-15 (`RectangularPattern.java`)

```
1 import java.util.Scanner;
2
3 /**
4  * This program displays a rectangular pattern
5  * of asterisks.
6 */
7
8 public class RectangularPattern
9 {
10    public static void main(String[] args)
11    {
12        int rows, cols;
13
14        // Create a Scanner object for keyboard input.
15        Scanner keyboard = new Scanner(System.in);
16
17        // Get the number of rows and columns.
18        System.out.print("How many rows? ");
19        rows = keyboard.nextInt();
20        System.out.print("How many columns? ");
21        cols = keyboard.nextInt();
22
23        for (int r = 0; r < rows; r++)
24        {
25            for (int c = 0; c < cols; c++)
26            {
27                System.out.print("*");
28            }
29            System.out.println();
30        }
31    }
```

```
32 }
```

Program Output

```
How many rows? 5 Enter
How many columns? 10 Enter
*****
*****
*****
*****
*****
*****
```

Let's look at another example. Suppose you want to print asterisks in a pattern that looks like the following triangle:

```
*
```

```
**
```

```
***
```

```
*** *
```

```
****
```

```
*****
```

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

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

Once again, think of the pattern as being arranged in rows and columns. The pattern has a total of eight rows. In the first row, there is one column. In the second row, there are two columns. In the third row, there are three columns. This continues to the eighth row, which has eight columns. [Code Listing 5-16](#) shows the program that produces this pattern.

Code Listing 5-16 (TrianglePattern.java)

```
1 import java.util.Scanner;
2
3 /**
4  * This program displays a triangle pattern.
5  */
6
7 public class TrianglePattern
```

```
8 {
9   public static void main(String[] args)
10  {
11    final int BASE_SIZE = 8;
12
13    for (int r = 0; r < BASE_SIZE; r++)
14    {
15      for (int c = 0; c < (r+1); c++)
16      {
17        System.out.print("*");
18      }
19      System.out.println();
20    }
21  }
22 }
```

Program Output

```
*
```

```
**
```

```
***
```

```
*** *
```

```
****
```

```
*****
```

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

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

The outer loop (which begins in line 13) will iterate eight times. As the loop iterates, the variable `r` will be assigned the values 0 through 7.

For each iteration of the outer loop, the inner loop will iterate $r + 1$ times. So,

- During the outer loop's 1st iteration, the variable `r` is assigned 0. The inner loop iterates one time, printing one asterisk.
- During the outer loop's 2nd iteration, the variable `r` is assigned 1. The inner loop iterates two times, printing two asterisks.
- During the outer loop's 3rd iteration, the variable `r` is assigned 2. The inner loop iterates three times, printing three asterisks.
- And so forth.

Let's look at another example. Suppose you want to display the following stairstep pattern:

```
#  
#  
#  
#  
#  
#
```

The pattern has six rows. In general, we can describe each row as having some number of spaces followed by a # character. Here's a row-by-row description:

First row: 0 spaces followed by a # character.

Second row: 1 space followed by a # character.

Third row: 2 spaces followed by a # character.

Fourth row: 3 spaces followed by a # character.

Fifth row: 4 spaces followed by a # character.

Sixth row: 5 spaces followed by a # character.

To display this pattern, we can write code containing a pair of nested loops that work in the following manner:

- The outer loop will iterate six times. Each iteration will perform the following:
 - The inner loop will display the correct number of spaces, side by side.
 - Then, a # character will be displayed.

[Code Listing 5-17](#) shows the Java code.

Code Listing 5-17 (StairStepPattern.java)

```

1 import java.util.Scanner;
2
3 /**
4  * This program displays a stairstep pattern.
5  */
6
7 public class StairStepPattern
8 {
9     public static void main(String[] args)
10    {
11        final int NUM_STEPS = 6;
12
13        for (int r = 0; r < NUM_STEPS; r++)
14        {
15            for (int c = 0; c < r; c++)
16            {
17                System.out.print(" ");
18            }
19            System.out.println("#");
20        }
21    }
22 }
```

Program Output

```

#
#
#
#
#
#
```

The outer loop (which begins in line 13) will iterate six times. As the loop iterates, the variable `r` will be assigned the values 0 through 5.

For each iteration of the outer loop, the inner loop will iterate `r` times. So,

- During the outer loop's 1st iteration, the variable `r` is assigned 0. The inner loop will not execute at this time.
- During the outer loop's 2nd iteration, the variable `r` is assigned 1. The inner loop iterates one time, printing one space.

- During the outer loop's 3rd iteration, the variable `r` is assigned 2. The inner loop iterates two times, printing two spaces.
- And so forth.

5.8 The break and continue Statements

Concept:

The `break` statement causes a loop to terminate early. The `continue` statement causes a loop to stop its current iteration and begin the next one.

The `break` statement, which was used with the `switch` statement in [Chapter 4](#), can also be placed inside a loop. When it is encountered, the loop stops, and the program jumps to the statement immediately following the loop. Although it is perfectly acceptable to use the `break` statement in a `switch` statement, it is considered “taboo” to use it in a loop. This is because it bypasses the normal condition required to terminate the loop, and it makes code difficult to understand and debug. For this reason, you should avoid using the `break` statement in a loop when possible.

The `continue` statement causes the current iteration of a loop to immediately end. When `continue` is encountered, all the statements in the body of the loop that appear after it are ignored, and the loop prepares for the next iteration. In a `while` loop, this means the program jumps to the boolean expression at the top of the loop. As usual, if the expression is still true, the next iteration begins. In a `do-while` loop, the program jumps to the boolean expression at the bottom of the loop, which determines whether the next iteration will begin. In a `for` loop, `continue` causes the update expression to be executed, and then the test expression is evaluated.

The `continue` statement should also be avoided. Like the `break` statement, it bypasses the loop’s logic, and makes the code difficult to understand and debug.

5.9 Deciding Which Loop to Use

Concept:

Although most repetitive algorithms can be written with any of the three types of loops, each works best in different situations.

Each of Java's three loops is ideal to use in different situations. Here's a short summary of when each loop should be used.

- The `while` loop. The `while` loop is a pretest loop. It is ideal in situations where you do not want the loop to iterate if the condition is `false` from the beginning. It is also ideal if you want to use a sentinel value to terminate the loop.
- The `do-while` loop. The `do-while` loop is a posttest loop. It is ideal in situations where you always want the loop to iterate at least once.
- The `for` loop. The `for` loop is a pretest loop that has built-in expressions for initializing, testing, and updating. These expressions make it very convenient to use a loop control variable as a counter. The `for` loop is ideal in situations where the exact number of iterations is known.

5.10 Introduction to File Input and Output

Concept:

The Java API provides several classes that you can use for writing data to a file and reading data from a file. To write data to a file, you can use the `PrintWriter` class, and optionally, the `FileWriter` class. To read data from a file, you can use the `Scanner` class, and the `File` class.

The programs you have written so far require you to reenter data each time the program runs. This is because the data stored in variables and objects in RAM disappear once the program stops running. To retain data between the times it runs, a program must have a way of saving the data.

Data may be saved in a file, which is usually stored on a computer's disk. Once the data is saved in a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time. In general, there are three steps taken when a file is used by a program:

1. The file must be *opened*. When the file is opened, a connection is established between the file and the program.
2. Data is then written to the file, or read from the file.
3. When the program is finished using the file, the file must be *closed*.

In this section, we will discuss how to write Java programs that read data from files and write data to files. The terms *input file* and *output file* are commonly used. An *input file* is a file from which a program reads data. It is called an input file because the data stored in it serves as input to the

program. An *output file* is a file to which a program writes data. It is called an output file because the program stores output in the file.

In general, there are two types of files: text and binary. A *text file* contains plain text and may be opened in a text editor such as Notepad. A *binary file* contains unformatted binary data, and you cannot view its contents with a text editor. In this chapter, we will discuss how to work with text files. Binary files will be discussed in [Chapter 10](#).

In this section, we will discuss a number of classes from the Java API that you will use to work with files. To use these classes, you will place the following `import` statement near the top of your program:

```
import java.io.*;
```

Using the PrintWriter Class to Write Data to a File

To write data to a file, you will create an instance of the `PrintWriter` class. The `PrintWriter` class allows you to open a file for writing. It also allows you to write data to the file using the same `print` and `println` methods you have been using to display data on the screen. You pass the name of the file that you wish to open, as a string, to the `PrintWriter` class's constructor. For example, the following statement creates a `PrintWriter` object, and passes the file name `StudentData.txt` to the constructor.

```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
```

This statement will create an empty file named `StudentData.txt` and establish a connection between it and the `PrintWriter` object referenced by `outputFile`. The file will be created in the current directory or folder.

You may also pass a reference to a `String` object as an argument to the `PrintWriter` constructor. For example, in the following code the user specifies the name of the file.

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter the filename: ");
String filename = keyboard.nextLine();
PrintWriter outputFile = new PrintWriter(filename);
```



Warning!

If the file that you are opening with the `PrintWriter` object already exists, it will be erased, then an empty file with the same name will be created.

Once you have created an instance of the `PrintWriter` class and opened a file, you can write data to the file using the `print` and `println` methods. You already know how to use `print` and `println` with `System.out` to display data on the screen. They are used the same way with a `PrintWriter` object to write data to a file. For example, assuming that `outputFile` references a `PrintWriter` object, the following statement writes the string "Jim" to the file.

```
outputFile.println("Jim");
```

When the program is finished writing data to the file, it must close the file. To close the file, use the `PrintWriter` class's `close` method. Here is an example of the method's use:

```
outputFile.close();
```

Your application should always close files when finished with them. This is because the system creates one or more buffers when a file is opened. A *buffer* is a small “holding section” of memory. When a program writes data to a file, that data is first written to the buffer. When the buffer is filled, all the information stored there is written to the file. This technique increases the system's performance because writing data to memory is faster than writing it to a disk. The `close` method writes any unsaved data remaining in the file buffer.

Once a file is closed, the connection between it and the `PrintWriter` object is removed. To perform further operations on the file, it must be opened again.

More about the PrintWriter Class's println Method

The PrintWriter class's `println` method writes a line of data to a file. For example, assume an application creates a file, and writes three students' first names and their test scores to the file with the following code.

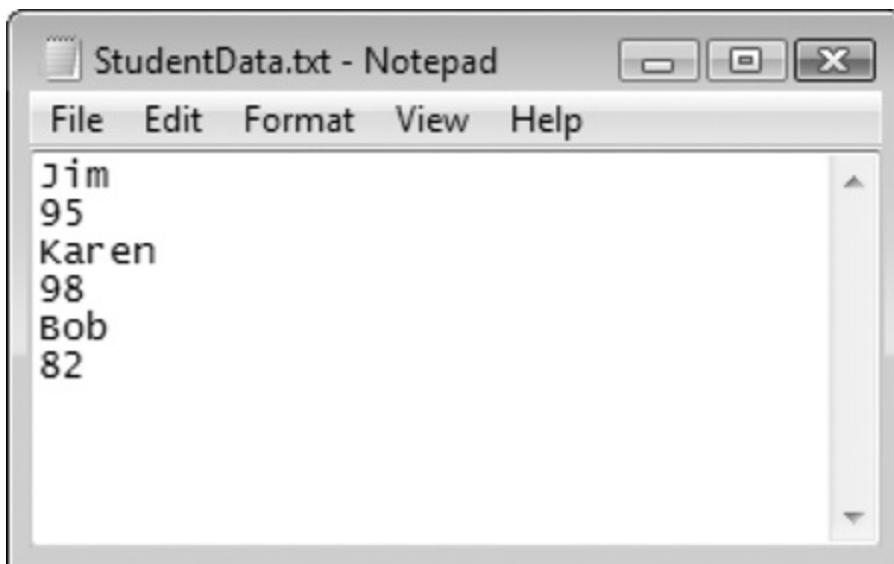
```
PrintWriter outputFile = new PrintWriter("StudentData.txt");
outputFile.println("Jim");
outputFile.println(95);
outputFile.println("Karen");
outputFile.println(98);
outputFile.println("Bob");
outputFile.println(82);
outputFile.close();
```

The `println` method writes data to the file, then writes a newline character immediately after the data. You can visualize the data written to the file in the following manner:

Jim<newline>95<newline>Karen<newline>98<newline>Bob<newline>82<ne

The newline characters are represented here as <*newline*>. You do not actually see the newline characters, but when the file is opened in a text editor such as Notepad, its contents will appear as shown in [Figure 5-11](#). As you can see from the figure, each newline character causes the data that follows it to be displayed on a new line.

Figure 5-11 File contents displayed in Notepad



[Figure 5-11 Full Alternative Text](#)

In addition to separating the contents of a file into lines, the newline character also serves as a delimiter. A *delimiter* is an item that separates other items. When you write data to a file using the `println` method, newline characters will separate the individual items of data. Later, you will see that the individual items of data in a file must be separated for them to be read from the file.

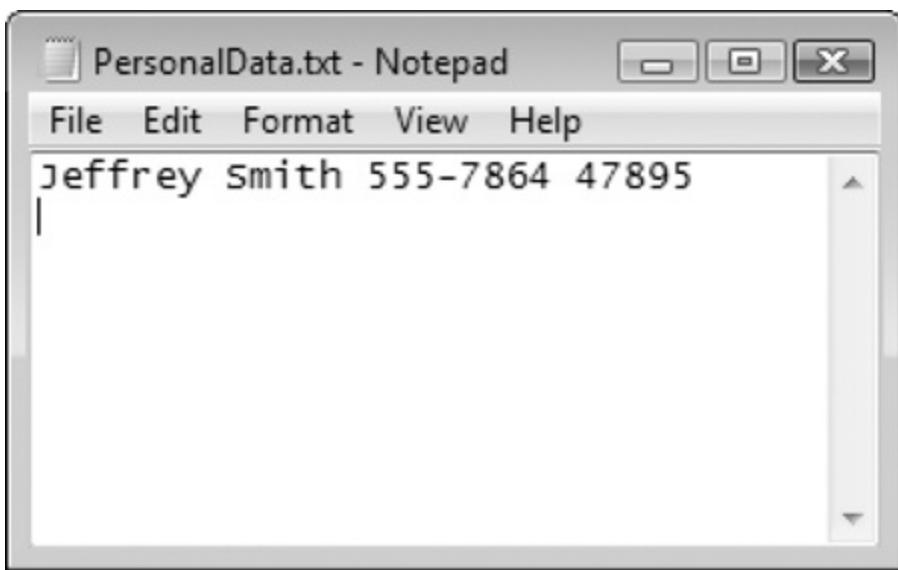
The PrintWriter Class's print Method

The `print` method is used to write an item of data to a file without writing the newline character. For example, look at the following code:

```
String name = "Jeffrey Smith";
String phone = "554-7864";
int idNumber = 47895;
PrintWriter outputFile = new PrintWriter("PersonalData.txt");
outputFile.print(name + " ");
outputFile.print(phone + " ");
outputFile.println(idNumber);
outputFile.close();
```

This code uses the `print` method to write the contents of the `name` object to the file, followed by a space (" "). Then it uses the `print` method to write the contents of the `phone` object to the file, followed by a space. Then, it uses the `println` method to write the contents of the `idNumber` variable, followed by a newline character. [Figure 5-12](#) shows the contents of the file displayed in Notepad.

Figure 5-12 Contents of file displayed in Notepad



[Figure 5-12 Full Alternative Text](#)

Adding a `throws` Clause to the Method Header

When an unexpected event occurs in a Java program, it is said that the program throws an exception. For now, you can think of an *exception* as a signal indicating that the program cannot continue until the unexpected event has been dealt with. For example, suppose you create a `PrintWriter` object

and pass the name of a file to its constructor. The `PrintWriter` object attempts to create the file, but unexpectedly the disk is full and the file cannot be created. Obviously, the program cannot continue until this situation has been dealt with, so an exception is thrown, which causes the program to suspend normal execution.

When an exception is thrown, the method that is executing must either deal with the exception, or throw it again. If the `main` method throws an exception, the program halts and an error message is displayed. Because `PrintWriter` objects are capable of throwing exceptions, we must either write code that deals with the possible exceptions, or allow our methods to rethrow the exceptions when they occur. In [Chapter 10](#), you will learn all about exceptions and how to respond to them, but for now, we will simply allow our methods to rethrow any exceptions that might occur.

To allow a method to rethrow an exception that has not been dealt with, you simply write a `throws` clause in the method header. The `throws` clause must indicate the type of exception that might be thrown. Here is an example:

```
public static void main(String[] args) throws IOException
```

This header indicates that the `main` method is capable of throwing an exception of the `IOException` type. This is the type of exception that `PrintWriter` objects are capable of throwing. So, any method that uses `PrintWriter` objects, and does not respond to their exceptions, must have this `throws` clause listed in its header.

In addition, any method that calls a method that uses a `PrintWriter` object should have a `throws IOException` clause in its header. For example, suppose the `main` method does not perform any file operations, but calls a method named `buildFile` that opens a file and writes data to it. Both the `buildFile` and `main` methods should have a `throws IOException` clause in their headers. Otherwise, a compiler error will occur.

An Example Program

Let's look at an example program that writes data to a file. The program in

[Code Listing 5-18](#) writes the names of your friends to a file.

Code Listing 5-18 (**FileWriteDemo.java**)

```
1 import java.util.Scanner; // Needed for Scanner
2 import java.io.*;      // Needed for PrintWriter and IOException
3
4 /**
5  * This program writes data to a file.
6  */
7
8 public class FilewriteDemo
9 {
10    public static void main(String[] args) throws IOException
11    {
12        String filename; // File name
13        String friendName; // Friend's name
14        int numFriends; // Number of friends
15
16        // Create a Scanner object for keyboard input.
17        Scanner keyboard = new Scanner(System.in);
18
19        // Get the number of friends.
20        System.out.print("How many friends do you have? ");
21        numFriends = keyboard.nextInt();
22
23        // Consume the remaining newline character.
24        keyboard.nextLine();
25
26        // Get the filename.
27        System.out.print("Enter the filename: ");
28        filename = keyboard.nextLine();
29
30        // Open the file.
31        PrintWriter outputFile = new PrintWriter(filename);
32
33        // Get data and write it to the file.
34        for (int i = 1; i <= numFriends; i++)
35        {
36            // Get the name of a friend.
37            System.out.print("Enter the name of friend "+
38                "number " + i + ": ");
```

```
39     friendName = keyboard.nextLine();
40
41     // Write the name to the file.
42     outputFile.println(friendName);
43 }
44
45     // Close the file.
46     outputFile.close();
47     System.out.println("Data written to the file.");
48 }
49 }
```

Program Output with Example Input Shown in Bold

```
How many friends do you have? 5 Enter
Enter the filename: MyFriends.txt Enter
Enter the name of friend number 1: Joe Enter
Enter the name of friend number 2: Rose Enter
Enter the name of friend number 3: Greg Enter
Enter the name of friend number 4: Kirk Enter
Enter the name of friend number 5: Renee Enter
Data written to the file.
```

The `import` statement in line 2 is necessary because this program uses the `PrintWriter` class, and because the `main` method header (in line 10) has a `throws IOException` clause. We need this clause in the `main` method header because objects of the `PrintWriter` class can potentially throw an `IOException`.

This program asks the user to enter the number of friends he or she has (in lines 20 and 21), then a name for the file that will be created (in lines 27 and 28). The `filename` variable references the name of the file, and is used in the following statement, in line 31:

```
PrintWriter outputFile = new PrintWriter(filename);
```

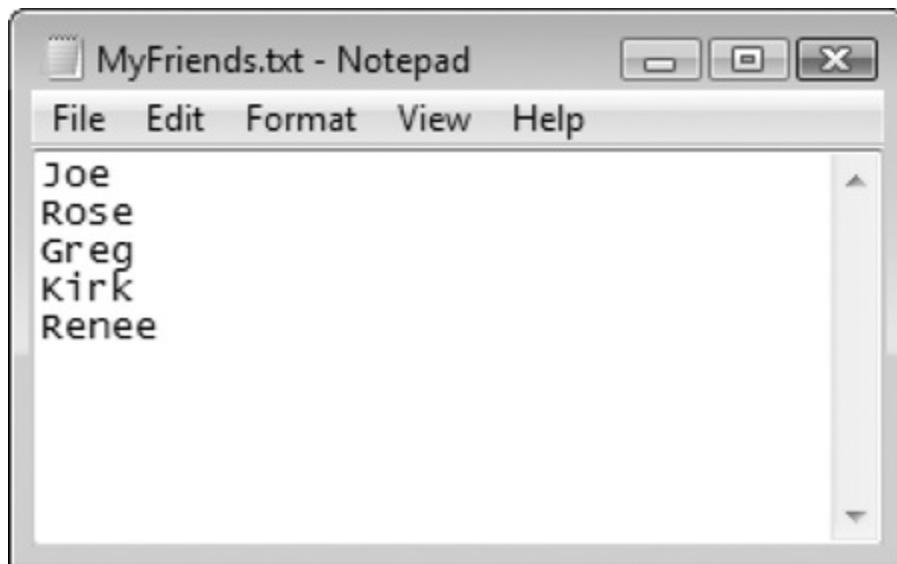
This statement opens the file and creates a `PrintWriter` object that can be used to write data to the file. The `for` loop in lines 34 through 43 performs an iteration for each friend that the user has, each time asking for the name of a

friend. The user's input is referenced by the `friendName` variable. Once the name is entered, it is written to the file with the following statement, which appears in line 42:

```
outputFile.println(friendName);
```

After the loop finishes, the file is closed in line 46. After the program is executed with the input shown in the example run, the file *MyFriends.txt* will be created. If we open the file in Notepad, we will see its contents as shown in [Figure 5-13](#).

Figure 5-13 Contents of the file displayed in Notepad



[Figure 5-13 Full Alternative Text](#)

Review

Before moving on, let's review the basic steps necessary when writing a program that writes data to a file:

1. You need the `import java.io.*;` statement in the top section of your program.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `PrintWriter` object must have a `throws IOException` clause in its header.
3. You create a `PrintWriter` object and pass the name of the file as a string to the constructor.
4. You use the `PrintWriter` class's `print` and `println` methods to write data to the file.
5. When finished writing to the file, you use the `PrintWriter` class's `close` method to close the file.

Appending Data to a File

When you pass the name of a file to the `PrintWriter` constructor and the file already exists, it will be erased and a new empty file with the same name will be created. Sometimes, however, you want to preserve an existing file and append new data to its current contents. Appending to a file means writing new data to the end of the data that already exists in the file.

To append data to an existing file, you first create an instance of the `FileWriter` class. You pass two arguments to the `FileWriter` constructor: a string containing the name of the file, and the boolean value `true`. Here is an example:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
```

This statement creates a `FileWriter` object and opens the file `MyFriends.txt` for writing. Any data written to the file will be appended to the file's existing contents. (If the file does not exist, it will be created.)

You still need to create a `PrintWriter` object so you can use the `print` and `println` methods to write data to the file. When you create the `PrintWriter`

object, you pass a reference to the `FileWriter` object as an argument to the `PrintWriter` constructor. For example, look at the following code:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
```

This creates a `PrintWriter` object that can be used to write data to the file *MyFriends.txt*. Any data written to the file will be appended to the file's existing contents. For example, assume the file *MyFriends.txt* exists and contains the following data:

```
Joe
Rose
Greg
Kirk
Renee
```

The following code opens the file and appends additional data to its existing contents:

```
FileWriter fwriter = new FileWriter("MyFriends.txt", true);
PrintWriter outputFile = new PrintWriter(fwriter);
outputFile.println("Bill");
outputFile.println("Steven");
outputFile.println("Sharon");
outputFile.close();
```

After this code executes, the *MyFriends.txt* file will contain the following data:

```
Joe
Rose
Greg
Kirk
Renee
Bill
Steven
Sharon
```



Note:

The `FileWriter` class also throws an `IOException` if the file cannot be opened for any reason.

Specifying the File Location

When you open a file, you may specify its path along with its file name. On a Windows computer, paths contain backslash characters. Remember that when a single backslash character appears in a string literal, it marks the beginning of an escape sequence such as "`\n`". Two backslash characters in a string literal represent a single backslash. So, when you provide a path in a string literal and the path contains backslash characters, you must use two backslash characters in the place of each single backslash character.

For example, the path "`E:\\Names.txt`" specifies that *Names.txt* is in the root folder of drive E:, and the path "`C:\\MyData\\Data.txt`" specifies that *Data.txt* is in the `\MyData` folder on drive C:. In the following statement, the file *Pricelist.txt* is created in the root folder of drive D:.

```
PrintWriter outputFile = new PrintWriter("D:\\PriceList.txt");
```

You only need to use double backslashes if the file's path is in a string literal. If your program asks the user to enter a path into a `String` object, which is then passed to the `PrintWriter` or `FileWriter` constructor, the user does not have to enter double backslashes.

Tip:

Java allows you to substitute forward slashes for backslashes in a Windows path. For example, the path "`C:\\MyData\\Data.txt`" could be written as "`C:/MyData/Data.txt`". This eliminates the need to use double backslashes.

On a UNIX or Linux computer, you can provide a path without any modifications. Here is an example:

```
PrintWriter outputFile = new PrintWriter("/home/rharrison/names.t
```

Reading Data from a File

In [Chapter 2](#), you learned how to use the `Scanner` class to read input from the keyboard. To read keyboard input, recall that we create a `Scanner` object, passing `System.in` to the `Scanner` class constructor. Here is an example:

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

Recall that the `System.in` object represents the keyboard. Passing `System.in` as an argument to the `Scanner` constructor specifies that the keyboard is the `Scanner` object's source of input.

You can also use the `Scanner` class to read input from a file. Instead of passing `System.in` to the `Scanner` class constructor, you pass a reference to a `File` object. Here is an example:

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

The first statement creates an instance of the `File` class. The `File` class is in the Java API, and is used to represent a file. Notice that we have passed the string "Customers.txt" to the constructor. This creates a `File` object that represents the file *Customers.txt*. In the second statement, we pass a reference to this `File` object as an argument to the `Scanner` class constructor. This creates a `Scanner` object that uses the file *Customers.txt* as its source of input. You can then use the same `Scanner` class methods that you learned about in [Chapter 2](#) to read items from the file. (See [Table 2-17](#) for a list of commonly used methods.) When you are finished reading from the file, you use the `Scanner` class's `close` method to close the file. For example, assuming the variable `inputFile` references a `Scanner` object, the following statement closes the file that is the object's source of input:

```
inputFile.close();
```

Reading Lines from a File with the

nextLine Method

The Scanner class's `nextLine` method reads a line of input and returns the line as a `String`. The program in [Code Listing 5-19](#) demonstrates how the `nextLine` method can be used to read a line from a file. This program asks the user to enter a file name, then displays the first line in the file on the screen.

Code Listing 5-19 (`ReadFirstLine.java`)

```
1 import java.util.Scanner; // Needed for Scanner
2 import java.io.*;      // Needed for File and IOException
3
4 /**
5  * This program reads the first line from a file.
6  */
7
8 public class ReadFirstLine
9 {
10    public static void main(String[] args) throws IOException
11    {
12        // Create a Scanner object for keyboard input.
13        Scanner keyboard = new Scanner(System.in);
14
15        // Get the file name.
16        System.out.print("Enter the name of a file: ");
17        String filename = keyboard.nextLine();
18
19        // Open the file.
20        File file = new File(filename);
21        Scanner inputFile = new Scanner(file);
22
23        // Read the first line from the file.
24        String line = inputFile.nextLine();
25
26        // Display the line.
27        System.out.println("The first line in the file is:");
28        System.out.println(line);
29
```

```
30     // Close the file.  
31     inputFile.close();  
32 }  
33 }
```

Program Output with Example Input Shown in Bold

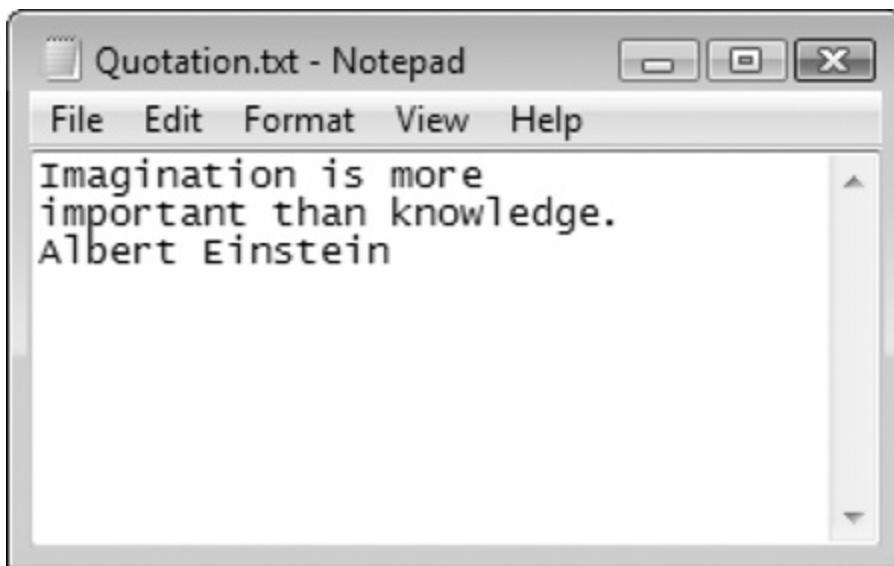
Enter the name of a file: **MyFriends.txt**
The first line in the file is:
Joe

This program gets the name of a file from the user in line 17. A `File` object is created in line 20 to represent the file, and a `Scanner` object is created in line 21 to read data from the file. Line 24 reads a line from the file. After this statement executes, the `line` variable references a `String` object holding the line that was read from the file. The line is displayed on the screen in line 28, and the file is closed in line 31.

Notice that this program creates two separate `Scanner` objects. The `Scanner` object created in line 13 reads data from the keyboard, and the `Scanner` object created in line 21 reads data from a file.

When a file is opened for reading, a special value known as a *read position* is internally maintained for that file. A file's read position marks the location of the next item that will be read from the file. When a file is opened, its read position is set to the first item in the file. When the item is read, the read position is advanced to the next item in the file. As subsequent items are read, the internal read position advances through the file. For example, consider the file *Quotation.txt*, shown in [Figure 5-14](#). As you can see from the figure, the file has three lines.

Figure 5-14 File with three lines



[Figure 5-14 Full Alternative Text](#)

You can visualize that the data is stored in the file in the following manner:

```
Imagination is more<newline>important than knowledge<newline>
Albert Einstein<newline>
```

Suppose a program opens the file with the following code.

```
File file = new File("Quotation.txt");
Scanner inputFile = new Scanner(file);
```

When this code opens the file, its read position is at the beginning of the first line, as illustrated in [Figure 5-15](#).

Figure 5-15 Initial read position

Read position —→ **Imagination is more
important than knowledge.
Albert Einstein**

[Figure 5-15 Full Alternative Text](#)

Now suppose the program uses the following statement to read a line from the file:

```
String str = inputFile.nextLine();
```

This statement will read a line from the file, beginning at the current read position. After the statement executes, the object referenced by str will contain the string “Imagination is more”. The file’s read position will be advanced to the next line, as illustrated in [Figure 5-16](#).

Figure 5-16 Read position after first line is read

Imagination is more
Read position → important than knowledge.
Albert Einstein

[Figure 5-16 Full Alternative Text](#)

If the nextLine method is called again, the second line will be read from the file, and the file’s read position will be advanced to the third line. After all the lines have been read, the read position will be at the end of the file.



Note:

The string returned from the nextLine method will not contain the newline character.

Adding a throws Clause to the

Method Header

When you pass a `File` object reference to the `Scanner` class constructor, the constructor will throw an exception of the `IOException` type if the specified file is not found. So, you will need to write a `throws IOException` clause in the header of any method that passes a `File` object reference to the `Scanner` class constructor.

Detecting the End of a File

Quite often a program must read the contents of a file without knowing the number of items stored in the file. For example, the `MyFriends.txt` file created by the program in [Code Listing 5-18](#) can have any number of names stored in it. This is because the program asks the user for the number of friends he or she has. If the user enters 5 for the number of friends, the program creates a file with five names in it. If the user enters 100, the program creates a file with 100 names in it.

The `Scanner` class has a method named `hasNext` that can be used to determine whether the file has more data that can be read. You call the `hasNext` method before you call any other methods to read from the file. If there is more data that can be read from the file, the `hasNext` method returns `true`. If the end of the file has been reached and there is no more data to read, the `hasNext` method returns `false`.

[Code Listing 5-20](#) shows an example. The program reads the file containing the names of your friends, which was created by the program in [Code Listing 5-18](#).

Code Listing 5-20 (`FileReadDemo.java`)

```
1 import java.util.Scanner; // Needed for Scanner
```

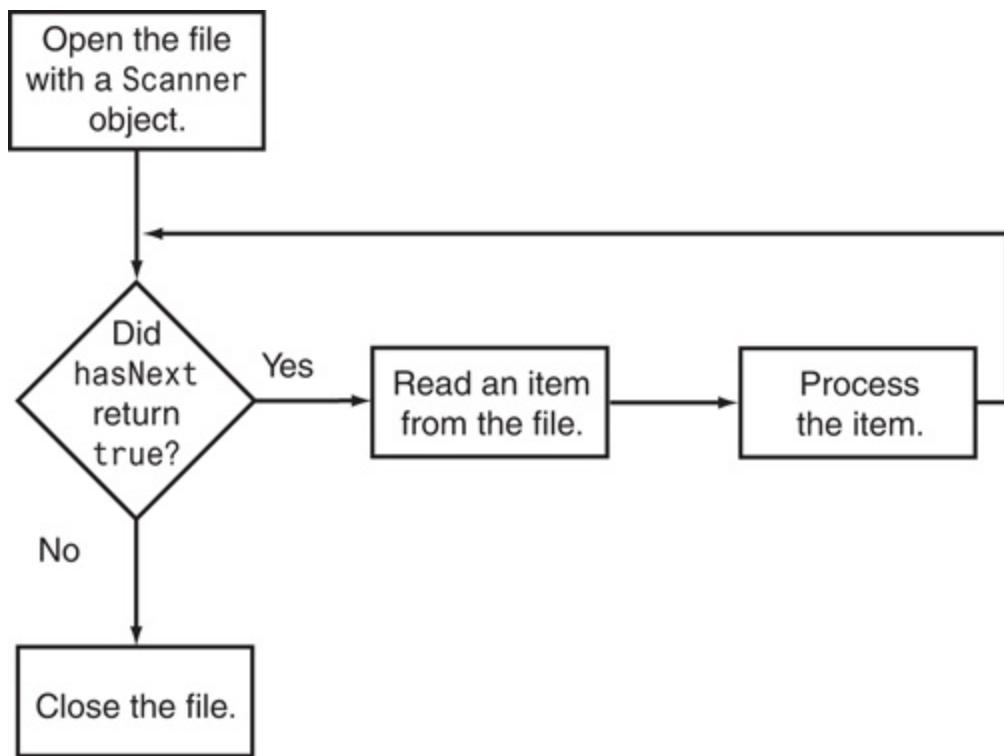
```
2 import java.io.*;           // Needed for File and IOException
3
4 /**
5  * This program reads data from a file.
6 */
7
8 public class FileReadDemo
9 {
10    public static void main(String[] args) throws IOException
11    {
12        // Create a Scanner object for keyboard input.
13        Scanner keyboard = new Scanner(System.in);
14
15        // Get the filename.
16        System.out.print("Enter the filename: ");
17        String filename = keyboard.nextLine();
18
19        // Open the file.
20        File file = new File(filename);
21        Scanner inputFile = new Scanner(file);
22
23        // Read lines from the file until no more are left.
24        while (inputFile.hasNext())
25        {
26            // Read the next name.
27            String friendName = inputFile.nextLine();
28
29            // Display the last name read.
30            System.out.println(friendName);
31        }
32
33        // Close the file.
34        inputFile.close();
35    }
36 }
```

Program Output with Example Input Shown in Bold

```
Enter the filename: MyFriends.txt 
Joe
Rose
Greg
Kirk
Renee
```

The file is opened, and a Scanner object to read it is created in line 21. The loop in lines 24 through 31 reads all of the lines from the file and displays them. In line 24, the loop calls the Scanner object's hasNext method. If the method returns true, then the file has more data to read. In that case, the next line is read from the file in line 27 and is displayed in line 30. The loop repeats until the hasNext method returns false in line 24. [Figure 5-17](#) shows the logic of reading a file until the end is reached.

Figure 5-17 Logic of reading a file until the end is reached



[Figure 5-17 Full Alternative Text](#)

Reading Primitive Values from a File

Recall from [Chapter 2](#) that the Scanner class provides methods for reading primitive values. These methods are named nextByte, nextDouble, nextFloat, nextInt, nextLong, and nextShort. [Table 2-18](#) gives more information on each of these methods, which can be used to read primitive values from a file. The FileSum class in [Code Listing 5-21](#) demonstrates how the nextDouble method can be used to read floating-point values from a file.

Code Listing 5-21 (FileSum.java)

```
1 import java.util.Scanner; // Needed for Scanner
2 import java.io.*;       // Needed for File and IOException
3
4 /**
5  * This class reads a series of numbers from a file and
6  * accumulates their sum.
7 */
8
9 public class FileSum
10 {
11     private double sum; // Accumulator
12
13     /**
14      * The constructor accepts a file name as its argument.
15      * The file is opened, the numbers read from it, and
16      * their sum is stored in the sum field.
17     */
18
19     public FileSum(String filename) throws IOException
20     {
21         String str; // To hold a line read from the file
22
23         // Create the necessary objects for file input.
24         File file = new File(filename);
25         Scanner inputFile = new Scanner(file);
26
27         // Initialize the accumulator.
28         sum = 0.0;
29
30         // Read all of the values from the file and
31         // calculate their total.
32         while (inputFile.hasNext())
33     {
34             // Read a value from the file.
```

```
35     double number = inputFile.nextDouble();
36
37     // Add the number to sum.
38     sum = sum + number;
39 }
40
41 // Close the file.
42 inputFile.close();
43 }
44
45 /**
46 * The getSum method returns the value in the sum field.
47 */
48
49 public double getSum()
50 {
51     return sum;
52 }
53 }
```

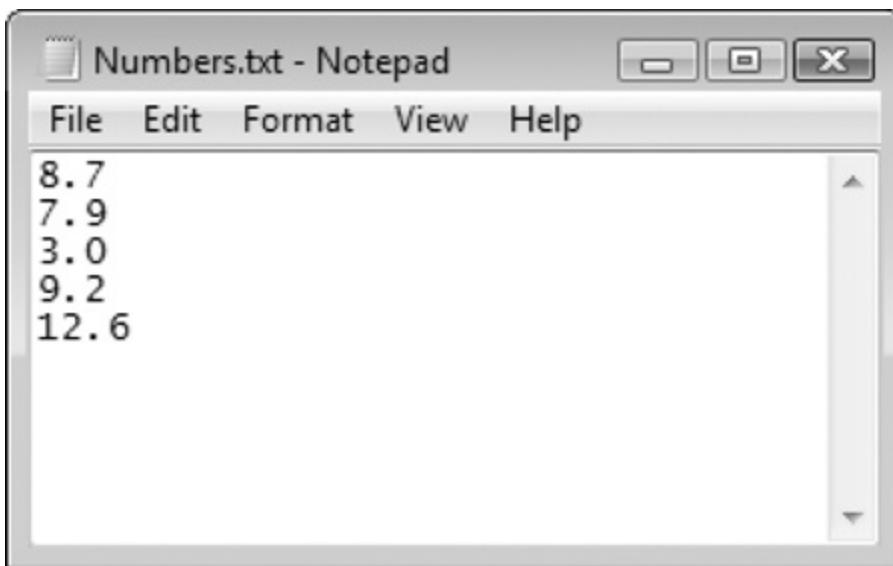
Program Output

The sum of the numbers in *Numbers.txt* is 41.4

The purpose of the `FileSum` class is to read the contents of a file that contains a series of numbers. The constructor, which begins at line 19, accepts a file name as its argument. The file is opened in line 25. The loop in lines 32 through 39 processes all of the numbers in the file. Line 35 reads a `double` and assigns it to the `number` variable. Line 38 adds `number` to the `sum` field. When this loop finishes, the `sum` field will contain the total of all the numbers read from the file. The `getSum` method, in lines 49 through 52, returns the value stored in the `sum` field.

To test this class, suppose the file *Numbers.txt* exists with the contents shown in [Figure 5-18](#). The program in [Code Listing 5-22](#) creates a `FileSum` object, passing the file's name, "Numbers.txt", to the constructor. The `getSum` method is then called to get the sum of the numbers.

Figure 5-18 Contents of *Numbers.txt* in Notepad



[Figure 5-18 Full Alternative Text](#)

Code Listing 5-22 (FileSumDemo.java)

```
1 import java.io.*; // Required for IOException
2
3 /**
4  * This program demonstrates the FileSum class.
5  */
6
7 public class FileSumDemo
8 {
9     public static void main(String[] args) throws IOException
10    {
11        // Create an instance of the FileSum class.
12        FileSum fs = new FileSum("Numbers.txt");
13
14        // Display the sum of the values in Numbers.txt.
15        System.out.println("The sum of the numbers in "+
16                           "Numbers.txt is "+
17                           fs.getSum());
18    }
19 }
```

Program Output

```
The sum of the numbers in Numbers.txt is 41.4
```

Review

Let's quickly review the steps necessary when writing a program that reads data from a file:

1. You will need the `import java.util.Scanner;` statement in the top section of your program, so you can use the `Scanner` class. You will also need the `import java.io.*;` statement in the top section of your program. This is required by the `File` class and the `IOException` class.
2. Because we have not yet learned how to respond to exceptions, any method that uses a `Scanner` object to open a file must have a `throws IOException` clause in its header.
3. You create a `File` object and pass the name of the file as a string to the constructor.
4. You create a `Scanner` object and pass a reference to the `File` object as an argument to the constructor.
5. You use the `Scanner` class's `nextLine` method to read a line from the file. The method returns the line of data as a string. To read primitive values, use methods such as `nextInt`, `nextDouble`, and so on.
6. Call the `Scanner` class's `hasNext` method to determine whether there is more data to read from the file. If the method returns `true`, then there is more data to read. If the method returns `false`, you have reached the end of the file.
7. When finished writing to the file, you use the `Scanner` class's `close` method to close the file.

Checking for a File's Existence

It's usually a good idea to make sure that a file exists before you try to open it for input. If you attempt to open a file for input and the file does not exist, the program will throw an exception and halt. For example, the program you saw in [Code Listing 5-20](#) will throw an exception at line 21 if the file being opened does not exist. Here is an example of the error message that will be displayed when this happens:

```
Exception in thread "main" java.io.FileNotFoundException: MyFriend
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(InputStream.java:106)
at java.util.Scanner.<init>(Scanner.java:636)
at FileReadDemo.main(FileReadDemo.java:21)
```

Rather than allowing the exception to be thrown and permitting this cryptic error message to be displayed, your program can check for the file's existence before it attempts to open the file. If the file does not exist, the program can display a more user-friendly error message, and gracefully shut down.

After you create a `File` object representing the file that you want to open, you can use the `File` class's `exists` method to determine whether the file exists. The method returns `true` if the file exists, or `false` if the file does not exist. [Code Listing 5-23](#) shows how to use the method. This is a modification of the `FileReadDemo` program in [Code Listing 5-20](#). This version of the program checks for the existence of the file before attempting to open it.

Code Listing 5-23 (`FileReadDemo2.java`)

```
1 import java.util.Scanner; // Needed for Scanner
2 import java.io.*;       // Needed for File and IOException
3
4 /**
5  * This program reads data from a file.
6  */
7
8 public class FileReadDemo2
9 {
10    public static void main(String[] args) throws IOException
```

```

11  {
12      // Create a Scanner object for keyboard input.
13      Scanner keyboard = new Scanner(System.in);
14
15      // Get the filename.
16      System.out.print("Enter the filename: ");
17      String filename = keyboard.nextLine();
18
19      // Make sure the file exists.
20      File file = new File(filename);
21      if (!file.exists())
22      {
23          // Display an error message.
24          System.out.println("The file " + filename +
25              " does not exist.");
26
27          // Exit the program.
28          System.exit(0);
29      }
30
31      // Open the file.
32      Scanner inputFile = new Scanner(file);
33
34      // Read lines from the file until no more are left.
35      while (inputFile.hasNext())
36      {
37          // Read the next name.
38          String friendName = inputFile.nextLine();
39
40          // Display the last name read.
41          System.out.println(friendName);
42      }
43
44      // Close the file.
45      inputFile.close();
46  }
47 }
```

Program Output (Assuming *badfile.txt* Does Not Exist)

Enter the filename: **badfile.txt** 
The file badfile.txt does not exist.

In line 20, the program creates a `File` object to represent the file. In line 21, the `if` statement calls the `file.exists()` method. Notice the use of the `!` operator. If the method returns `false`, indicating that the file does not exist,

the code in lines 23 through 28 executes. The statement in lines 24 and 25 displays an error message, and line 28 calls the `System.exit(0)` method, which shuts the program down.

The previous example shows you how to make sure that a file exists before trying to open it for input. But, when you are opening a file for output, sometimes you want to make sure the file does *not* exist. When you use a `PrintWriter` object to open a file, the file will be deleted if it already exists. If you do not want to erase the existing file, you have to check for its existence before creating the `PrintWriter` object. [Code Listing 5-24](#) shows you how to use the `File` class's `exists` method in this type of situation. This is a modification of the program you saw in [Code Listing 5-18](#).

Code Listing 5-24 (`FileWriteDemo2.java`)

```
1 import java.io.*;      // Needed for File and IOException
2 import java.util.Scanner; // Needed for Scanner
3
4 /**
5  * This program writes data to a file. It makes sure the
6  * specified file does not exist before opening it.
7 */
8
9 public class FileWriteDemo2
10 {
11     public static void main(String[] args) throws IOException
12     {
13         String filename;    // File name
14         String friendName; // Friend's name
15         int numFriends;    // Number of friends
16
17         // Create a Scanner object for keyboard input.
18         Scanner keyboard = new Scanner(System.in);
19
20         // Get the number of friends.
21         System.out.print("How many friends do you have? ");
22         numFriends = keyboard.nextInt();
23
24         // Consume the remaining newline character.
25         keyboard.nextLine();
```

```

26
27     // Get the filename.
28     System.out.print("Enter the filename: ");
29     filename = keyboard.nextLine();
30
31     // Make sure the file does not exist.
32     File file = new File(filename);
33     if (file.exists())
34     {
35         // Display an error message.
36         System.out.println("The file " + filename +
37             " already exists.");
38
39         // Exit the program.
40         System.exit(0);
41     }
42
43     // Open the file.
44     PrintWriter outputFile = new PrintWriter(file);
45
46     // Get data and write it to the file.
47     for (int i = 1; i <= numFriends; i++)
48     {
49         // Get the name of a friend.
50         System.out.print("Enter the name of friend " +
51             "number " + i + ": ");
52         friendName = keyboard.nextLine();
53
54         // Write the name to the file.
55         outputFile.println(friendName);
56     }
57
58     // Close the file.
59     outputFile.close();
60     System.out.println("Data written to the file.");
61 }
62 }
```

Program Output with Example Input Shown in Bold

How many friends do you have? **2** 
 Enter the filename: **MyFriends.txt** 
 The file MyFriends.txt already exists.

Line 32 creates a `File` object representing the file. The `if` statement in line

33 calls the `file.exists()` method. If the method returns `true`, then the file exists. In this case, the code in lines 35 through 40 executes. This code displays an error message and shuts the program down. If the file does not exist, the rest of the program executes.

Notice that in line 44 we pass a reference to the `File` object to the `PrintWriter` constructor. In previous programs that created an instance `PrintWriter`, we passed a file name to the constructor. If you have a reference to a `File` object that represents the file you wish to open, as we do in this program, you have the option of passing it to the `PrintWriter` constructor.



Checkpoint

1. 5.16 What is the difference between an input file and an output file?
2. 5.17 What `import` statement will you need in a program that performs file operations?
3. 5.18 What class do you use to write data to a file?
4. 5.19 Write code that does the following: Opens a file named `MyName.txt`, writes your first name to the file, then closes the file.
5. 5.20 What classes do you use to read data from a file?
6. 5.21 Write code that does the following: Opens a file named `MyName.txt`, reads the first line from the file and displays it, then closes the file.
7. 5.22 You are opening an existing file for output. How do you open the file without erasing it, and at the same time, make sure that new data written to the file is appended to the end of the file's existing data?
8. 5.23 What clause must you write in the header of a method that performs a file operation?

See the Amortization Class Case Study for an in-depth example using this chapter's topics. The case study is available on this book's online resource page at www.pearsonhighered.com/gaddis.

5.11 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- Using the increment or decrement operator in the wrong mode. When the increment or decrement operator is placed in front of (to the left of) its operand, it is used in prefix mode. When either of these operators are placed behind (to the right of) their operand, they are used in postfix mode.
- Forgetting to enclose the boolean expression in a `while` loop or a `do-while` loop inside parentheses.
- Placing a semicolon at the end of a `while` or `for` loop's header. When you write a semicolon at the end of a `while` or `for` loop's header, Java assumes that the conditionally executed statement is a null or empty statement. This usually results in an infinite loop.
- Forgetting to write the semicolon at the end of the `do-while` loop. The `do-while` loop must be terminated with a semicolon.
- Forgetting to enclose multiple statements in the body of a loop in braces. Normally, a loop conditionally executes only one statement. To conditionally execute more than one statement, you must place the statements in braces.
- Using commas instead of semicolons to separate the initialization, test, and update expressions in a `for` loop.
- Forgetting to write code in the body of a `while` or `do-while` loop that modifies the loop control variable. If a `while` or `do-while` loop's boolean expression never becomes `false`, the loop will repeat indefinitely. You must have code in the body of the loop that modifies the loop control variable so the boolean expression will at some point become `false`.

- Using a sentinel value that can also be a valid data value. Remember, a sentinel is a special value that cannot be mistaken as a member of a list of data items, and signals that there are no more data items from the list to be processed. If you choose as a sentinel a value that might also appear in the list, the loop will prematurely terminate if it encounters the value in the list.
- Forgetting to initialize an accumulator to zero. For an accumulator to keep a correct running total, it must be initialized to zero before any values are added to it.

Review Questions and Exercises

Multiple Choice and True/False

1. What will the `println` statement in the following program segment display?

```
int x = 5;  
System.out.println(x++);
```

1. 5
2. 6
3. 0
4. None of these

2. What will the `println` statement in the following program segment display?

```
int x = 5;  
System.out.println(++x);
```

1. 5
2. 6
3. 0
4. None of these

3. In the expression `number++`, the `++` operator is in what mode?

1. prefix

2. pretest
 3. postfix
 4. posttest
4. What is each repetition of a loop known as?
1. cycle
 2. revolution
 3. orbit
 4. iteration
5. This is a variable that controls the number of iterations performed by a loop.
1. loop control variable
 2. accumulator
 3. iteration register variable
 4. repetition meter
6. The `while` loop is this type of loop.
1. pretest
 2. posttest
 3. prefix
 4. postfix
7. The `do-while` loop is this type of loop.
1. pretest

2. posttest
 3. prefix
 4. postfix
8. The `for` loop is this type of loop.
1. pretest
 2. posttest
 3. prefix
 4. postfix
9. This type of loop has no way of ending and repeats until the program is interrupted.
1. indeterminate
 2. interminable
 3. infinite
 4. timeless
10. This type of loop always executes at least one time.
1. `while`
 2. `do-while`
 3. `for`
 4. Any of these
11. This expression is executed by the `for` loop only once, regardless of the number of iterations.

1. initialization expression
 2. test expression
 3. update expression
 4. preincrement expression
12. This is a variable that keeps a running total.
1. sentinel
 2. sum
 3. total
 4. accumulator
13. This is a special value that signals when there are no more items from a list of items to be processed. This value cannot be mistaken as an item from the list.
1. sentinel
 2. flag
 3. signal
 4. accumulator
14. To open a file for writing, you use the following class:
1. PrintWriter
 2. FileOpen
 3. outputFile
 4. FileReader

15. To open a file for reading, you use the following classes:
1. File and Writer
 2. File and Output
 3. File and Input
 4. File and Scanner
16. When a program is finished using a file, it should do this.
1. Erase the file
 2. Close the file
 3. Throw an exception
 4. Reset the read position
17. This class allows you to use the `print` and `println` methods to write data to a file.
1. File
 2. FileReader
 3. outputFile
 4. PrintWriter
18. This class allows you to read a line from a file.
1. FileWriter
 2. Scanner
 3. inputFile
 4. FileReader

19. True or False: The `while` loop is a pretest loop.
20. True or False: The `do-while` loop is a pretest loop.
21. True or False: The `for` loop is a posttest loop.
22. True or False: It is not necessary to initialize accumulator variables.
23. True or False: One limitation of the `for` loop is that only one variable may be initialized in the initialization expression.
24. True or False: A variable may be defined in the initialization expression of the `for` loop.
25. True or False: In a nested loop, the inner loop goes through all of its iterations for every single iteration of the outer loop.
26. True or False: To calculate the total number of iterations of a nested loop, add the number of iterations of all the loops.

Find the Error

Find the errors in the following code.

```
1. // This code contains ERRORS!
// It adds two numbers entered by the user.
int num1, num2;
String input;
char again;
Scanner keyboard = new Scanner(System.in);
while (again == 'y' || again == 'Y')
    System.out.print("Enter a number: ");
    num1 = keyboard.nextInt();
    System.out.print("Enter another number: ");
    num2 = keyboard.nextInt();
    System.out.println("Their sum is "+ (num1 + num2));
    System.out.println("Do you want to do this again? ");
    keyboard.nextLine(); // Consume remaining newline
    input = keyboard.nextLine();
    again = input.charAt(0);
```

```

2. // This code contains ERRORS!
int count = 1, total;
while (count <= 100)
    total += count;
System.out.print("The sum of the numbers 1 - 100 is ");
System.out.println(total);

3. // This code contains ERRORS!
Scanner keyboard = new Scanner(System.in);
int choice, num1, num2;
do
{
    System.out.print("Enter a number: ");
    num1 = keyboard.nextInt();
    System.out.print("Enter another number: ");
    num2 = keyboard.nextInt();
    System.out.println("Their sum is " + (num1 + num2));
    System.out.println("Do you want to do this again? ");
    System.out.print("1 = yes, 0 = no ");
    choice = keyboard.nextInt();
} while (Choice = 1)

4. // This code contains ERRORS!
// Print the numbers 1 through 10.
for (int count = 1, count <= 10, count++)
{
    System.out.println(count);
    count++;
}

```

Algorithm Workbench

1. Write a `while` loop that lets the user enter a number. The number should be multiplied by 10 and the result stored in the variable `product`. The loop should iterate as long as `product` contains a value less than 100.
2. Write a `do-while` loop that asks the user to enter two numbers. The numbers should be added and the sum displayed. The user should be asked if he or she wishes to perform the operation again. If so, the loop should repeat. Otherwise, it should terminate.

3. Write a `for` loop that displays the following set of numbers:

0, 10, 20, 30, 40, 50 . . . 1000

4. Write a loop that asks the user to enter a number. The loop should iterate 10 times and keep a running total of the numbers entered.
5. Write a `for` loop that calculates the total of the following series of numbers:

130+229+328+...301

6. Write a nested loop that displays 10 rows of '#' characters. There should be 15 '#' characters in each row.
7. Write nested loops to draw this pattern:

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

8. Write nested loops to draw this pattern:

```
##  
# #  
# #  
# #  
# #  
# #
```

9. Complete the following program so that it performs the following actions 10 times:

- Generates a random number that is either 0 or 1.
- Displays either the word “Yes” or the word “No” depending on the

random number that was generated.

```
// Write the necessary import statement(s) here.
public class ReviewQuestion
{
    public static void main(String[] args)
    {
        // Write the necessary code here.
    }
}
```

10. Convert the `while` loop in the following code segment to a `do-while` loop:

```
Scanner keyboard = new Scanner(System.in);
int x = 1;
while (x > 0)
{
    System.out.print("Enter a number: ");
    x = keyboard.nextInt();
}
```

11. Convert the `do-while` loop in the following code segment to a `while` loop:

```
Scanner keyboard = new Scanner(System.in);
String input;
char sure;
do
{
    System.out.print("Are you sure you want to quit? ");
    input = keyboard.next();
    sure = input.charAt(0);
} while (sure != 'Y' && sure != 'N');
```

12. Convert the `while` loop in the following code segment to a `for` loop:

```
int count = 0;
while (count < 50)
{
    System.out.println("count is " + count);
    count++;
}
```

13. Convert the following `for` loop to a `while` loop:

```
for (int x = 50; x > 0; x--)
{
    System.out.println(x + " seconds to go.");
}
```

14. Write an input validation loop that asks the user to enter a number in the range of 1 through 5.
15. Write an input validation loop that asks the user to enter the words “yes” or “no”.
16. Write code that does the following: Opens a file named *numberList.txt*, uses a loop to write the numbers 1 through 100 to the file, then closes the file.
17. Write code that does the following: Opens the *numberList.txt* file created by the code in Question 13, reads all of the numbers from the file and displays them, then closes the file.
18. Modify the code you wrote in Question 14 so it adds all of the numbers read from the file and displays their total.
19. Write code that opens a file named *numberList.txt* for writing, but does not erase the file’s contents if it already exists.

Short Answer

1. Briefly describe the difference between the prefix and postfix modes used by the increment and decrement operators.
2. Why should you indent the statements in the body of a loop?
3. Describe the difference between pretest loops and posttest loops.
4. Why are the statements in the body of a loop called conditionally executed statements?

5. Describe the difference between the `while` loop and the `do-while` loop.
6. Which loop should you use in situations where you wish the loop to repeat until the Boolean expression is false, and the loop should not execute if the test expression is false to begin with?
7. Which loop should you use in situations where you wish the loop to repeat until the Boolean expression is false, but the loop should execute at least one time?
8. Which loop should you use when you know the number of required iterations?
9. Why is it critical that accumulator variables be properly initialized?
10. What is an infinite loop? Write the code for an infinite loop.
11. Describe a programming problem that would require the use of an accumulator.
12. What does it mean to let the user control a loop?
13. What is the advantage of using a sentinel?
14. Why must the value chosen for use as a sentinel be carefully selected?
15. Describe a programming problem requiring the use of nested loops.
16. How does a file buffer increase a program's performance?
17. Why should a program close a file when finished using it?
18. What is a file's read position? Where is the read position when a file is first opened for reading?
19. When writing data to a file, what is the difference between the `print` and the `println` methods?
20. What does the `Scanner` class's `hasNext` method return when the end of

the file has been reached?

21. What is a potential error that can occur when a file is opened for reading?
22. What does it mean to append data to a file?
23. How do you open a file so that new data will be written to the end of the file's existing data?

Programming Challenges

1. Sum of Numbers

Write a program that asks the user for a positive nonzero integer value. The program should use a loop to get the sum of all the integers from 1 up to the number entered. For example, if the user enters 50, the loop will find the sum of 1, 2, 3, 4, . . . , 50.

2. Distance Traveled

The distance a vehicle travels can be calculated as follows:

$$\text{Distance} = \text{Speed} * \text{Time}$$

For example, if a train travels 40 miles per hour (mph) for three hours, the distance traveled is 120 miles.

Design a class that stores the speed of a vehicle (in miles per hour) and the number of hours it has traveled. It should have a method named `getDistance` that returns the distance, in miles, that the vehicle has traveled.

Demonstrate the class in a program that uses a loop to display the distance a vehicle has traveled for each hour of a time period specified by the user. For example, if a vehicle is traveling at 40 mph for a three-hour time period, it should display a report similar to the one shown here.

Hour	Distance Traveled
1	40
2	80
3	120

Hour	Distance Traveled
1	40
2	80
3	120

Input Validation: Do not accept a negative number for speed, and do not accept any value less than one for time traveled.

3. Distance File

Modify the program you wrote for Programming Challenge 2 (Distance Traveled) so that it writes the report to a file instead of the screen. Open the file in Notepad or another text editor to confirm the output.

4. Pennies for Pay

Write a program that calculates how much a person would earn over a period of time if his or her salary is one penny the first day, two pennies the second day, and continues to double each day. The program should display a table showing the salary for each day, then show the total pay at the end of the period. The output should be displayed in a dollar amount, not the number of pennies.



VideoNote The Pennies for Pay Problem

Input Validation: Do not accept a number less than 1 for the number of days worked.

5. Hotel Occupancy

A hotel's occupancy rate is calculated as follows:

Occupancy rate = number of rooms occupied ÷ total number of rooms

Write a program that calculates the occupancy rate for each floor of a hotel. The program should start by asking for the number of floors the hotel has. A loop should then iterate once for each floor. During each iteration, the loop should ask the user for the number of rooms on the floor, and how many of them are occupied. After all the iterations, the program should display the number of rooms the hotel has, the number that are occupied, the number that are vacant, and the occupancy rate for the hotel.

Input Validation: Do not accept a value less than 1 for the number of floors. Do not accept a number less than 10 for the number of rooms on

a floor.

6. Population

Write a class that will predict the size of a population of organisms. The class should store the starting number of organisms, their average daily population increase (as a percentage), and the number of days they will multiply. The class should have a method that uses a loop to display the size of the population for each day.

Test the class in a program that asks the user for the starting size of the population, their average daily increase, and the number of days they will multiply. The program should display the daily population.

Input Validation: Do not accept a number less than 2 for the starting size of the population. Do not accept a negative number for average daily population increase. Do not accept a number less than 1 for the number of days they will multiply.

7. Average Rainfall

Write a program that uses nested loops to collect data and calculate the average rainfall over a period of years. The program should first ask for the number of years. The outer loop will iterate once for each year. The inner loop will iterate 12 times, once for each month. Each iteration of the inner loop will ask the user for the inches of rainfall for that month.

After all iterations, the program should display the number of months, the total inches of rainfall, and the average rainfall per month for the entire period.

Input Validation: Do not accept a number less than 1 for the number of years. Do not accept negative numbers for the monthly rainfall.

8. The Greatest and Least of These

Write a program with a loop that lets the user enter a series of integers. The user should enter -99 to signal the end of the series. After all the

numbers have been entered, the program should display the largest and smallest numbers entered.

9. Payroll Report

Design a `Payroll` class that stores an employee's ID number, gross pay, state tax, federal tax, and FICA withholdings. The class should have a method that calculates the employee's net pay, as follows:

$$\text{net pay} = \text{gross pay} - \text{state tax} - \text{federal tax} - \text{FICA withholdings}$$

Use the class in a program that displays a weekly payroll report. A loop in the program should ask the user for the employee ID number, gross pay, state tax, federal tax, and FICA withholdings, and should pass these values to an instance of the `Payroll` class. The net pay should be displayed. The loop should terminate when 0 is entered for the employee number. After the data is entered, the program should display totals for gross pay, state tax, federal tax, FICA withholdings, and net pay.

Input Validation: Do not accept negative numbers for any of the items entered. Do not accept values for state, federal, or FICA withholdings that are greater than the gross pay. If the state tax + federal tax + FICA withholdings for any employee are greater than gross pay, print an error message, and ask the user to reenter the data for that employee.

10. SavingsAccount Class

Design a `SavingsAccount` class that stores a savings account's annual interest rate and balance. The class constructor should accept the amount of the savings account's starting balance. The class should also have methods for subtracting the amount of a withdrawal, adding the amount of a deposit, and adding the amount of monthly interest to the balance. The monthly interest rate is the annual interest rate divided by 12. To add the monthly interest to the balance, multiply the monthly interest rate by the balance and add the result to the balance.

Test the class in a program that calculates the balance of a savings account at the end of a period of time. The program should ask the user

for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should then iterate once for every month, performing the following:

1. Ask the user for the amount deposited into the account during the month. Use the class method to add this amount to the account balance.
2. Ask the user for the amount withdrawn from the account during the month. Use the class method to subtract this amount from the account balance.
3. Use the class method to calculate the monthly interest.

After the last iteration, the program should display the ending balance, the total amount of deposits, the total amount of withdrawals, and the total interest earned.

11. Deposit and Withdrawal Files

Use Notepad or another text editor to create a text file named *Deposits.txt*. The file should contain the following numbers, one per line:

100.00
125.00
78.92
37.55

Next, create a text file named *Withdrawals.txt*. The file should contain the following numbers, one per line:

29.88
110.00
27.52
50.00
12.90

The numbers in the *Deposits.txt* file are the amounts of deposits that were made to a savings account during the month, and the numbers in

the *Withdrawals.txt* file are the amounts of withdrawals that were made during the month. Write a program that creates an instance of the *SavingsAccount* class that you wrote in Programming Challenge 10. The starting balance for the object is 500.00. The program should read the values from the *Deposits.txt* file and use the object's method to add them to the account balance. The program should read the values from the *Withdrawals.txt* file and use the object's method to subtract them from the account balance. The program should call the class method to calculate the monthly interest, then display the ending balance and the total interest earned.

12. Bar Chart

Write a program that asks the user to enter today's sales for five stores. The program should then display a bar chart comparing each store's sales. Create each bar in the bar chart by displaying a row of asterisks. Each asterisk should represent \$100 of sales. Here is an example of the program's output.

```
Enter today's sales for store 1: 1000 Enter
Enter today's sales for store 2: 1200 Enter
Enter today's sales for store 3: 1800 Enter
Enter today's sales for store 4: 800 Enter
Enter today's sales for store 5: 1900 Enter
SALES BAR CHART
Store 1: *****
Store 2: ******
Store 3: *****
Store 4: *****
Store 5: *****
```

13. Centigrade to Fahrenheit Table

Write a program that displays a table of the centigrade temperatures 0 through 20 and their Fahrenheit equivalents. The formula for converting a temperature from centigrade to Fahrenheit is

$$F = \frac{9}{5}C + 32$$

where F is the Fahrenheit temperature and C is the centigrade temperature. Your program must use a loop to display the table.

14. FileDisplay Class

Write a class named `FileDisplay` with the following methods:

- Constructor. The class’s constructor should take the name of a file as an argument.
- `displayHead`. This method should display only the first five lines of the file’s contents. If the file contains less than five lines, it should display the file’s entire contents.
- `displayContents`. This method should display the entire contents of the file, the name of which was passed to the constructor.
- `displayWithLineNumbers`. This method should display the contents of the file, the name of which was passed to the constructor. Each line should be preceded with a line number followed by a colon. The line numbering should start at 1.

15. UppercaseFile Class

Write a class named `UppercaseFile`. The class’s constructor should accept two file names as arguments. The first file should be opened for reading and the second file should be opened for writing. The class should read the contents of the first file, change all characters to uppercase, and store the results in the second file. The second file will be a copy of the first file, except all the characters will be uppercase. Use Notepad or another text editor to create a simple file that can be used to test the class.

16. Budget Analysis

Write a program that asks the user to enter the amount that he or she has budgeted for a month. A loop should then prompt the user to enter each of his or her expenses for the month, and keep a running total. When the loop finishes, the program should display the amount that the user is over or under budget.

17. Random Number Guessing Game

Write a program that generates a random number and asks the user to guess what the number is. If the user's guess is higher than the random number, the program should display "Too high, try again." If the user's guess is lower than the random number, the program should display "Too low, try again." The program should use a loop that repeats until the user correctly guesses the random number.

18. Random Number Guessing Game Enhancement

Enhance the program that you wrote for Programming Challenge 17 so that it keeps a count of the number of guesses that the user makes. When the user correctly guesses the random number, the program should display the number of guesses.

19. Square Display

Write a program that asks the user for a positive integer no greater than 15. The program should then display a square on the screen using the character 'x'. The number entered by the user will be the length of each side of the square. For example, if the user enters 5, the program should display the following:

```
xxxxx  
xxxxx  
xxxxx  
xxxxx  
xxxxx
```

If the user enters 8, the program should display the following:

```
xxxxxxxx  
xxxxxxxx
```

```
XXXXXXX  
XXXXXXX  
XXXXXXX  
XXXXXXX  
XXXXXXX  
XXXXXXX
```

20. Coin-Toss Simulator

Write a class named `Coin`. The `Coin` class should have the following field:

- A `String` named `sideUp`. The `sideUp` field will hold either “heads” or “tails” indicating the side of the coin that is facing up.

The `Coin` class should have the following methods:

- A no-arg constructor that calls the `toss` method, described next.
- A `void` method named `toss` that simulates the tossing of the coin. When the `toss` method is called, it generates a random number in the range of 0 through 1. If the random number is 0, then it sets the `sideUp` field to “heads”. If the random number is 1, then it sets the `sideUp` field to “tails”.
- A method named `getSideUp` that returns the value of the `sideUp` field.

Write a program that demonstrates the `Coin` class. The program should create an instance of the class and display the side that is initially facing up. Then, use a loop to toss the coin 20 times. Each time the coin is tossed, display the side that is facing up. The program should keep count of the number of times a head is facing up, and the number of times a tail is facing up, then display those values after the loop finishes.

21. Tossing Coins for a Dollar

For this assignment, you will create a game program using the `Coin` class from Programming Challenge 20. The program should have three instances of the `Coin` class: one representing a quarter, one representing

a dime, and one representing a nickel.

When the game begins, your starting balance is \$0. During each round of the game, the program will toss the simulated coins. When a coin is tossed, the value of the coin is added to your balance if it lands heads-up. For example, if the quarter lands heads-up, 25 cents is added to your balance. Nothing is added to your balance for coins that land tails-up. The game is over when your balance reaches one dollar or more. If your balance is exactly one dollar, you win the game. You lose if your balance exceeds one dollar.

22. Dice Game

Write a program that uses the `Die` class that was presented in [Chapter 4](#) to play a simple dice game between the computer and the user. The program should create two instances of the `Die` class (each a six-sided die). One `Die` object is the computer's die, and the other `Die` object is the user's die.

The program should have a loop that iterates 10 times. Each time the loop iterates, it should roll both dice. The die with the highest value wins. (In case of a tie, there is no winner for that particular roll of the dice.)

As the loop iterates, the program should keep count of the number of times the computer wins, and the number of times that the user wins. After the loop performs all of its iterations, the program should display who was the grand winner, the computer or the user.

23. Fishing Game Simulation

Write a program that uses the `Die` class that was presented in [Chapter 4](#) to simulate a fishing game. In this game, a six-sided die is rolled to determine what the user has caught. Each possible item is worth a certain number of fishing points. The points will remain hidden until the user is finished fishing, then a message is displayed congratulating the user depending on the number of fishing points gained.

Here are some suggestions for the game's design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player wants to fish for more items.
- At the beginning of each round, the program will ask the user whether or not they want to continue fishing.
- The program simulates the rolling of a six-sided die. (Use the `Die` class that was demonstrated in [Chapter 4](#).)
- Each item that can be caught is represented by a number generated from the die. For example, 1 for “a huge fish”, 2 for “an old shoe”, 3 for “a little fish”, and so on.
- Each item the user catches is worth a different amount of points.
- The loop keeps a running total of the user’s fishing points.
- After the loop has finished, the total number of fishing points is displayed, along with a message that varies depending on the number of points earned.

24. A Game of 21

Write a program that uses the `Die` class that was presented in [Chapter 4](#) to write a program that lets the user play against the computer in a variation of the popular blackjack card game. In this variation of the game, two six-sided dice are used instead of cards. The dice are rolled, and the player tries to beat the computer’s hidden total without going over 21.

Here are some suggestions for the game’s design:

- Each round of the game is performed as an iteration of a loop that repeats as long as the player agrees to roll the dice, and the player’s total does not exceed 21.
- At the beginning of each round, the program will ask the user

whether he or she wants to roll the dice to accumulate points.

- During each round, the program simulates the rolling of two six-sided dice. It rolls the dice first for the computer, then it asks the user if he or she wants to roll. (Use the `Die` class that was demonstrated in [Chapter 4](#) to simulate the dice).
- The loop keeps a running total of both the computer and the user’s points.
- The computer’s total should remain hidden until the loop has finished.
- After the loop has finished, the computer’s total is revealed, and whoever the player with the most points without going over 21 wins.

25. ESP Game

Write a program that tests your ESP (extrasensory perception). The program should randomly select the name of a color from the following list of words:

- *Red, Green, Blue, Orange, Yellow*

To select a word, the program can generate a random number. For example, if the number is 0, the selected word is *Red*; if the number is 1, the selected word is *Green*; and so forth.

Next, the program should ask the user to enter the color that the computer has selected. After the user has entered his or her guess, the program should display the name of the randomly selected color. The program should repeat this 10 times then display the number of times the user correctly guessed the selected color.

26. Slot Machine Simulation

A slot machine is a gambling device into which the user inserts money, then pulls a lever (or presses a button). The slot machine then displays a

set of random images. If two or more of the images match, the user wins an amount of money that the slot machine dispenses back to the user.

Create a program that simulates a slot machine. When the program runs, it should do the following:

- Ask the user to enter the amount of money he or she wants to enter into the slot machine.
- Instead of displaying images, have the program randomly select a word from the following list:
 - *Cherries, Oranges, Plums, Bells, Melons, Bars*

To select a word, the program can generate a random number in the range of 0 through 5. If the number is 0, the selected word is *Cherries*; if the number is 1, the selected word is *Oranges*; and so forth. The program should randomly select a word from this list three times and display all three of the words.

- If none of the randomly selected words match, the program informs the user that he or she has won \$0. If two of the words match, the program informs the user that he or she has won two times the amount entered. If three of the words match, the program informs the user that he or she has won three times the amount entered.
- The program asks if the user wants to play again. If so, these steps are repeated. If not, the program displays the total amount of money entered into the slot machine, and the total amount won.

27. Personal Web Page Generator

Write a program that asks the user for his or her name, then asks the user to enter a sentence that describes him or herself. Here is an example of the program's screen:

```
Enter your name: Julie Taylor 
Describe yourself: I am a computer science major, a member of
```

Enter

Once the user has entered the requested input, the program should create an HTML file, containing the input, for a simple Web page. Here is an example of the HTML content, using the sample input previously shown:

```
<html>
<head>
</head>
<body>
    <center>
        <h1>Julie Taylor</h1>
    </center>
    <hr />
    I am a computer science major, a member of the Jazz club,
    and I hope to work as a mobile app developer after I gradua
    <hr />
</body>
</html>
```

Chapter 6 A Second Look at Classes and Objects

Topics

1. [6.1 Static Class Members](#)
2. [6.2 Overloaded Methods](#)
3. [6.3 Overloaded Constructors](#)
4. [6.4 Passing Objects as Arguments to Methods](#)
5. [6.5 Returning Objects from Methods](#)
6. [6.6 The `toString` Method](#)
7. [6.7 Writing an `equals` Method](#)
8. [6.8 Methods That Copy Objects](#)
9. [6.9 Aggregation](#)
10. [6.10 The `this` Reference Variable](#)
11. [6.11 Inner Classes](#)
12. [6.12 Enumerated Types](#)
13. [6.13 Garbage Collection](#)
14. [6.14 Focus on Object-Oriented Design: Object Collaboration](#)
15. [6.15 Common Errors to Avoid](#)

6.1 Static Class Members

Concept:

A static class member belongs to the class, not objects instantiated from the class.

A Quick Review of Instance Fields and Instance Methods

Recall from [Chapter 3](#) that each instance of a class has its own set of fields, which are known as instance fields. You can create several instances of a class and store different values in each instance's fields. For example, the `Rectangle` class we created in [Chapter 3](#) has a `length` field and a `width` field. Let's say that `box` references an instance of the `Rectangle` class, and execute the following statement:

```
box.setLength(10);
```

This statement stores the value 10 in the `length` field that belongs to the instance referenced by `box`. You can think of instance fields as belonging to an instance of a class.

You will also recall that classes may have instance methods as well. When you call an instance method, it performs an operation on a specific instance of the class. For example, assuming that `box` references an instance of the `Rectangle` class, look at the following statement:

```
x = box.getLength();
```

This statement calls the `getLength` method, which returns the value of the `length` field that belongs to a specific instance of the `Rectangle` class, the one referenced by `box`. Both instance fields and instance methods are associated with a specific instance of a class, and they cannot be used until an instance of the class is created.

Static Members

It is possible to create a field or method that does not belong to any instance of a class. Such members are known as *static fields* and *static methods*. When a value is stored in a static field, it is not stored in an instance of the class. In fact, an instance of the class doesn't even have to exist for values to be stored in the class's static fields. Likewise, static methods do not operate on the fields that belong to any instance of the class. Instead, they can operate only on static fields. In this section, we will take a closer look at static members. First, we will examine static fields.

Static Fields

When a field is declared `static`, there will be only one copy of the field in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static field is shared by all instances of the class. For example, the `Countable` class shown in [Code Listing 6-1](#) uses a static field to keep count of the number of instances of the class that are created.

Code Listing 6-1 (`Countable.java`)

```
1  /**
2   * This class demonstrates a static field.
3   */
4
5 public class Countable
6 {
7     private static int instanceCount = 0;
8 }
```

```

9     /**
10      * The constructor increments the static
11      * field instanceCount. This keeps track
12      * of the number of instances of this
13      * class that are created.
14      */
15
16     public Countable()
17     {
18         instanceCount++;
19     }
20
21     /**
22      * The getInstanceCount method returns
23      * the value in the instanceCount field,
24      * which is the number of instances of
25      * this class that have been created.
26      */
27
28     public int getInstanceCount()
29     {
30         return instanceCount;
31     }
32 }
```

First, notice the declaration of the static field named `instanceCount` in line 7. A static field is created by placing the key word `static` after the access specifier, and before the field's data type. Notice that we have explicitly initialized the `instanceCount` field with the value 0. This initialization only takes place once, regardless of the number of instances of the class that are created.



Note:

Java automatically stores 0 in all uninitialized numeric static member variables. The `instanceCount` field in this class is explicitly initialized, so it is clear to anyone reading the code that the field starts with the value 0.

Next, look at the constructor which appears in lines 16 through 19. The statement in line 18 uses the `++` operator to increment the `instanceCount` field. Each time an instance of the `Countable` class is created, the constructor

will be called, and the `instanceCount` field will be incremented. As a result, the `instanceCount` field will contain the number of instances of the `Countable` class that have been created. The `getInstanceCount` method, which appears in lines 28 through 31, can be used to retrieve this value. The program in [Code Listing 6-2](#) demonstrates this class.

Code Listing 6-2 (StaticDemo.java)

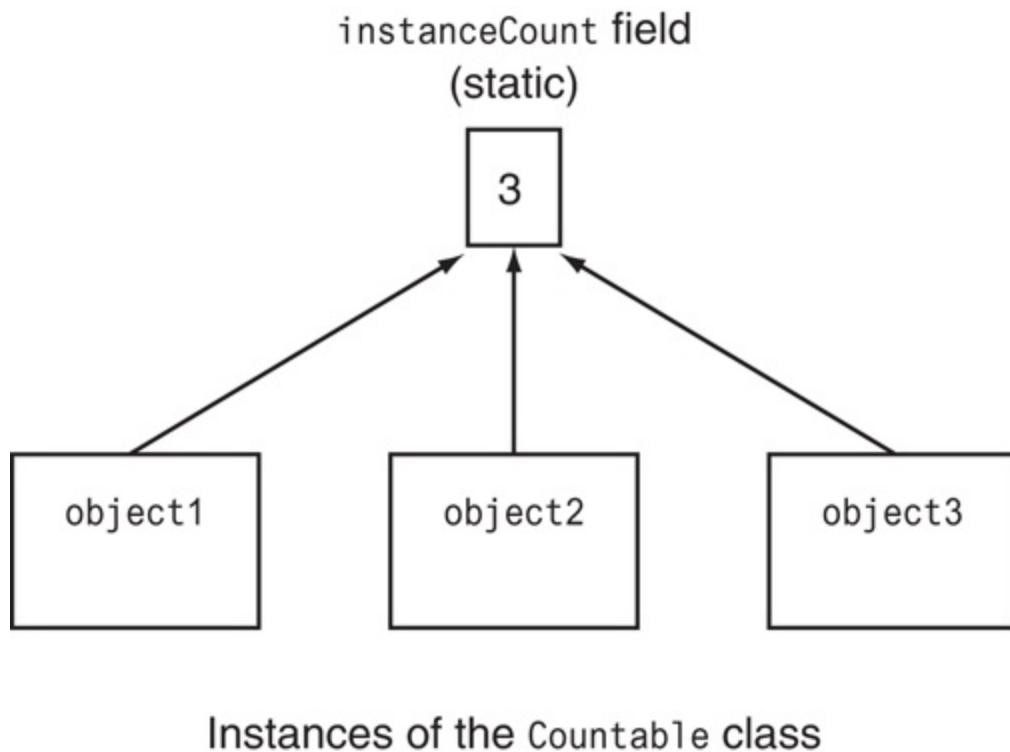
```
1  /**
2   * This program demonstrates the Countable class.
3   */
4
5  public class StaticDemo
6  {
7      public static void main(String [] args)
8      {
9          int objectCount;
10
11         // Create three instances of the
12         // Countable class.
13         Countable object1 = new Countable();
14         Countable object2 = new Countable();
15         Countable object3 = new Countable();
16
17         // Get the number of instances from
18         // the class's static field.
19         objectCount = object1.getInstanceCount();
20         System.out.println(objectCount + " instances " +
21                           "of the class were created.");
22     }
23 }
```

Program Output

3 instances of the class were created.

The program creates three instances of the `Countable` class, referenced by the variables `object1`, `object2`, and `object3`. Although there are three instances of the class, there is only one copy of the static field. This is illustrated in [Figure 6-1](#).

Figure 6-1 All instances of the class share the static field



[Figure 6-1 Full Alternative Text](#)

In line 19, the program calls the `getInstanceCount` method to retrieve the number of instances that have been created. Although the program calls the `getInstanceCount` method from `object1`, the same value would be returned from any of the objects.

Static Methods

When a class contains a static method, it isn't necessary for an instance of the class to be created to execute the method. The program in [Code Listing 6-3](#) shows an example of a class with static methods.

Code Listing 6-3 (Metric.java)

```
1  /**
2   * This class demonstrates static methods.
3   */
4
5  public class Metric
6  {
7      /**
8       * The milesToKilometers method converts miles
9       * to kilometers. A distance in miles should be
10      * passed into the miles parameter. The method
11      * returns the equivalent distance in kilometers.
12      */
13
14     public static double milesToKilometers(double miles)
15     {
16         return miles * 1.609;
17     }
18
19     /**
20      * The kilometersToMiles method converts kilometers
21      * to miles. A distance in kilometers should be
22      * passed into the kilometers parameter. The method
23      * returns the equivalent distance in miles.
24      */
25
26     public static double kilometersToMiles(double kilometers)
27     {
28         return kilometers / 1.609;
29     }
30 }
```

A static method is created by placing the key word `static` after the access specifier in the method header. The `Metric` class has two static methods: `milesToKilometers` and `kilometersToMiles`. Because they are declared as `static`, they belong to the class, and may be called without any instances of the class being in existence. You simply write the name of the class before the dot operator in the method call. Here is an example:

```
kilos = Metric.milesToKilometers(10.0);
```

This statement calls the `milesToKilometers` method, passing the value `10.0`

as an argument. Notice that the method is not called from an instance of the class, but is called directly from the `Metric` class. [Code Listing 6-4](#) shows a program that uses the `Metric` class.

Code Listing 6-4 (`MetricDemo.java`)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates the Metric class.
5  */
6
7 public class MetricDemo
8 {
9     public static void main(String[] args)
10    {
11        double miles, // A distance in miles
12            kilos; // A distance in kilometers
13
14        // Create a Scanner object for keyboard input.
15        Scanner keyboard = new Scanner(System.in);
16
17        // Get a distance in miles.
18        System.out.print("Enter a distance in miles: ");
19        miles = keyboard.nextDouble();
20
21        // Convert the distance to kilometers.
22        kilos = Metric.milesToKilometers(miles);
23        System.out.printf("%.2f miles equals %.2f kilometers.\n"
24                           "                miles, kilos);
25
26        // Get a distance in kilometers.
27        System.out.print("Enter a distance in kilometers: ");
28        kilos = keyboard.nextDouble();
29
30        // Convert the distance to kilometers.
31        miles = Metric.kilometersToMiles(kilos);
32        System.out.printf("%.2f kilometers equals %.2f miles.\n"
33                           "                kilos, miles);
34    }
35 }
```

Program Output with Example Input Shown in Bold

```
Enter a distance in miles: 10 Enter  
10.00 miles equals 16.09 kilometers.  
Enter a distance in kilometers: 100 Enter  
100.00 kilometers equals 62.15 miles.
```

Static methods are convenient for many tasks because they can be called directly from the class, as needed. They are most often used to create utility classes that perform operations on data, but have no need to collect and store data. The `Metric` class is a good example. It is used as a container to hold methods that convert miles to kilometers and vice versa, but is not intended to store any data.

If a class has both static members and instance members, keep the following points in mind:

- An instance method can refer to a static variable in the same class. You saw this demonstrated in the `Countable` class in [Code Listing 6-1](#).
- An instance method can call a static method.
- It is not necessary to create an object to execute a static method. However, an instance method or an instance variable can be referred to only in the context of an object. Because a static method can be executed without an object of the class being in existence, a static method cannot refer to an instance variable or an instance method of the same class—that is, unless the static method has an object reference. For example, a static method could create an object of the class, then use the object reference to call instance methods or refer to instance variables.

The Math Class

The Java `Math` class is a collection of static methods for performing specific mathematical operations. In [Chapter 2](#), you were introduced to the `Math.pow` method, which returns the value of a number raised to a power. For example, the following statement raises 5 to the 10th power and assigns the result to the `result` variable:

```
result = Math.pow(5.0, 10.0);
```

The `Math` class also has a method named `sqrt` that returns the square root of its argument. For example, in the following statement the `Math.sqrt` method returns the square root of the value stored in the `number` variable, and assigns the result to the `result` variable:

```
result = Math.sqrt(number);
```

The `Math.sqrt` method accepts a `double` argument, and returns a `double`. The `Math` class also has a static final variable named `PI`, which is set to the mathematical constant pi, or π . It is defined as 3.14159265358979323846. The following statement uses `Math.PI` in a calculation.

```
circumference = Math.PI * diameter;
```



Note:

The `Math` class has other static members. For more information, see Appendix F, available on this book's online resource page at www.pearsonhighered.com/gaddis.



Checkpoint

1. 6.1 What is the difference between an instance member variable and a static member variable?
2. 6.2 What action is possible with a static method that isn't possible with an instance method?
3. 6.3 Describe the limitation of static methods.

6.2 Overloaded Methods

Concept:

Two or more methods in a class may have the same name as long as their signatures are different.

Sometimes, you will need to perform an operation in different ways, perhaps using items of different data types. For example, consider the following two methods, `squareInt` and `squareDouble`.

```
public static int squareInt(int number)
{
    return number * number;
}
public static double squareDouble(double number)
{
    return number * number;
}
```

Both of these methods accept an argument and return the square of that argument. The `squareInt` method accepts an `int`, and the `squareDouble` method accepts a `double`. Although this approach will work, a better solution is to use method overloading. In *method overloading*, multiple methods have the same name, but use different parameters.

For example, [Code Listing 6-5](#) shows the `MyMath` class. This class has two methods named `square`. Both methods do the same thing: return the square of their argument. One version of the method accepts an `int` argument, and the other accepts a `double`.

Code Listing 6-5 (`MyMath.java`)

```
1  /**
2   * This class overloads the square method.
3   */
4
5  public class MyMath
6  {
7      public static int square(int number)
8      {
9          return number * number;
10     }
11
12     public static double square(double number)
13     {
14         return number * number;
15     }
16 }
```

The program in [Code Listing 6-6](#) uses both square methods.

Code Listing 6-6 (OverloadingDemo.java)

```
1  import java.util.Scanner;
2
3  /**
4   * This program uses the MyMath class to
5   * demonstrate overloaded methods.
6   */
7
8  public class OverloadingDemo
9  {
10     public static void main(String[] args)
11     {
12         int iNumber;
13         double dNumber;
14
15         // Create a Scanner object for keyboard input.
16         Scanner keyboard = new Scanner(System.in);
17
18         // Get an integer and display its square.
19         System.out.print("Enter an integer: ");
20         iNumber = keyboard.nextInt();
21         System.out.println("That number's square is " +
```

```

22                     MyMath.square(iNumber));
23
24     // Get a double and display its square.
25     System.out.print("Enter a double: ");
26     dNumber = keyboard.nextDouble();
27     System.out.println("That number's square is " +
28                         MyMath.square(dNumber));
29 }
30 }
```

Program Output with Example Input Shown in Bold

```

Enter an integer: 5 
That number's square is 25
Enter a double: 1.2 
That number's square is 1.44
```

The process of matching a method call with the correct method is known as *binding*. When an overloaded method is being called, Java uses the method's name and parameter list to determine which method to bind the call to. In [Code Listing 6-6](#), when an `int` argument is passed to `square`, the version of the method that has an `int` parameter is called. Likewise, when a `double` argument is passed to `square`, the version with a `double` parameter is called.

Method Signatures

Java uses a method's signature to distinguish it from other methods of the same name. A method's *signature* consists of the method's name and the data types of the method's parameters, in the order that they appear. For example, here are the signatures of the `square` methods that appear in the `MyMath` class:

```

square(int)
square(double)
```

Note that the method's return type is not part of the signature. For example, the `square` method cannot be overloaded in the following manner:

```

public static int square(int number)
{
    return number * number;
```

```

}

// ERROR! The following method's parameter list does
// not differ from the previous square method.

public static double square(int number)
{
    return number * number;
}

```

Although these methods have different return values, their signatures are the same. For this reason, an error message will be issued when a class containing these methods is compiled.

Overloading is also convenient when there are similar methods that use a different number of parameters. For example, [Code Listing 6-7](#) shows the Pay class, which uses two methods, each named `getWeeklyPay`. These methods return an employee's gross weekly pay. One version of the method returns the weekly pay for an hourly paid employee. It uses an `int` parameter for the number of hours worked, and a `double` parameter for the hourly pay rate. The other version of the method returns the weekly pay for a salaried employee. It uses a `double` parameter for the yearly salary. [Code Listing 6-8](#) illustrates the use of the Pay class.

Code Listing 6-7 (Pay.java)

```

1  /**
2   * This class uses overloaded methods to return an employee's
3   * weekly salary.
4  */
5
6 public class Pay
7 {
8     /**
9      * The following method calculates the gross weekly pay of
10     * an hourly paid employee. The parameter hours holds the
11     * number of hours worked. The parameter payRate holds the
12     * hourly pay rate. The method returns the weekly salary.
13     */
14
15    public static double getWeeklyPay(int hours, double payRat

```

```

16     {
17         return hours * payRate;
18     }
19
20 /**
21  * The following method overloads the getWeeklyPay method.
22  * It calculates the gross weekly pay of a salaried
23  * employee. The parameter holds the employee's yearly
24  * salary. The method returns the weekly salary.
25 */
26
27 public static double getWeeklyPay(double yearlySalary)
28 {
29     return yearlySalary / 52;
30 }
31 }
```

Code Listing 6-8 (WeeklyPay.java)

```

1 import java.util.Scanner;
2
3 /**
4  * This program uses the Pay class to determine an
5  * employee's weekly pay. It can process hourly paid
6  * or salaried employees.
7 */
8
9 public class WeeklyPay
10 {
11     public static void main(String[] args)
12     {
13         String selection; // The user's selection, H or S
14         int hours; // The number of hours worked
15         double hourlyRate; // The hourly pay rate
16         double yearly; // The yearly salary
17
18         // Create a Scanner object for keyboard input.
19         Scanner keyboard = new Scanner(System.in);
20
21         // Determine whether the employee is hourly paid or sal
22         System.out.println("Do you want to calculate the " +
23                           "weekly salary of an hourly-paid");
24         System.out.println("or a salaried employee?");
25         System.out.print("Enter H for hourly or S for salaried:");
26         selection = keyboard.nextLine();
```

```

27
28     // Determine and display the weekly pay.
29     switch(selection.charAt(0))
30     {
31         case 'H':
32         case 'h':
33             System.out.print("How many hours were worked? ");
34             hours = keyboard.nextInt();
35             System.out.print("What is the hourly pay rate? ")
36             hourlyRate = keyboard.nextDouble();
37             System.out.printf("The weekly gross pay is $%, .2f
38                               Pay.getWeeklyPay(hours, hourlyR
39             break;
40
41         case 'S':
42         case 's':
43             System.out.print("What is the annual salary? ");
44             yearly = keyboard.nextDouble();
45             System.out.printf("The weekly gross pay is $%, .2f
46                               Pay.getWeeklyPay(yearly));
47             break;
48
49         default:
50             System.out.println("Invalid selection.");
51     }
52 }
53 }
```

Program Output with Example Input Shown in Bold

Do you want to calculate the weekly salary of an hourly paid or a salaried employee?

Enter H for hourly or S for salaried: **h** 

How many hours were worked? **40** 

What is the hourly pay rate? **25.00** 

The weekly gross pay is \$1,000.00

Program Output with Example Input Shown in Bold

Do you want to calculate the weekly salary of an hourly paid or a salaried employee?

Enter H for hourly or S for salaried: **s** 

What is the annual salary? **65000.00** The weekly gross pay is \$1,250.00

6.3 Overloaded Constructors

Concept:

More than one constructor may be defined for a class.

A class's constructor may be overloaded in the same manner as other methods. The rules for overloading constructors are the same for overloading other methods: Each version of the constructor must have a different signature. As long as each constructor has a unique signature, the compiler can tell them apart. For example, recall the `Rectangle` class from [Chapter 3](#).

We added a constructor to the class that accepts two arguments, which are assigned to the `length` and `width` fields. [Code Listing 6-9](#) shows how the class can be modified with the addition of another constructor.

Code Listing 6-9 (`Rectangle.java`)

```
1  /**
2   * Rectangle class
3   */
4
5  public class Rectangle
6  {
7      private double length;
8      private double width;
9
10     /**
11      * Constructor
12      */
13
14     public Rectangle()
15     {
16         length = 0.0;
```

```
17         width = 0.0;
18     }
19
20    /**
21     * Overloaded constructor
22     */
23
24    public Rectangle(double len, double w)
25    {
26        length = len;
27        width = w;
28    }
```

The first constructor accepts no arguments, and assigns 0.0 to the length and width fields. The second constructor accepts two arguments that are assigned to the length and width fields. The program in [Code Listing 6-10](#) demonstrates both of these constructors.

Code Listing 6-10

(TwoRectangles.java)

```

22                     box1.getWidth());
23
24     // Create another Rectangle object and use
25     // the second constructor.
26
27     box2 = new Rectangle(5.0, 10.0);
28     System.out.println("The box2 object's length " +
29                         "and width are " +
30                         box2.getLength() + " and " +
31                         box2.getWidth());
32 }
33 }
```

Program Output

The box1 object's length and width are 0.0 and 0.0
 The box2 object's length and width are 5.0 and 10.0

This program declares two Rectangle variables, box1 and box2. The statement in line 18 creates the first Rectangle object. Because the statement passes no arguments to the constructor, the first constructor is executed:

```

public Rectangle()
{
    length = 0.0;
    width = 0.0;
}
```

This is verified by a call to `System.out.println` in lines 19 through 22, which displays the contents of the `length` and `width` fields as 0.0. Here is the statement in line 27, which creates the second Rectangle object:

```
box2 = new Rectangle(5.0, 10.0);
```

Because this statement passes two double arguments to the constructor, the second constructor is called:

```

public Rectangle(double len, double w)
{
    length = len;
    width = w;
}
```

The call to `System.out.println` statement (in lines 28 through 31) verifies

that the content of the `length` field is 5.0, and the `width` field is 10.0.

The Default Constructor Revisited

Recall from [Chapter 3](#) that if you do not write a constructor for a class, Java automatically provides one. The constructor Java provides is known as the default constructor. It sets all of the class's numeric fields to 0 and boolean fields to `false`. If the class has any fields that are reference variables, the default constructor sets all of them to the value `null`, which means they do not reference anything.

Java provides a default constructor only when you do not write any constructors for a class. If a class has a constructor that accepts arguments, but it does not also have a no-arg constructor (a constructor that does not accept arguments), you cannot create an instance of the class without passing arguments to the constructor. Therefore, any time you write a constructor for a class, you should also write a no-arg constructor if you want to be able to create instances of the class without passing arguments to the constructor.

The `InventoryItem` Class

Let's look at a class that uses multiple constructors. The `InventoryItem` class holds simple data about an item in an inventory. A description of the item is stored in the `description` field, and the number of units on hand is stored in the `units` field. [Figure 6-2](#) shows a UML diagram for the class.

Figure 6-2 UML diagram for the `InventoryItem` class

InventoryItem
- description : String - units : int
+ InventoryItem() + InventoryItem(d : String) + InventoryItem(d : String, u : int) + setDescription(d : String) : void + setUnits(u : int) : void + getDescription() : String + getUnits() : int

[Figure 6-2 Full Alternative Text](#)

The code for the class is shown in [Code Listing 6-11](#).

Code Listing 6-11 (InventoryItem.java)

```

1  /**
2   * This class uses three constructors.
3   */
4
5  public class InventoryItem
6  {
7      private String description; // Item description
8      private int units;         // Units on-hand
9
10     /**
11      * No-arg constructor
12     */
13
14     public InventoryItem()
15     {
16         description = "";
17         units = 0;
18     }
19
20     /**

```

```
21     * The following constructor accepts a
22     * String argument that is assigned to the
23     * description field.
24     */
25
26     public InventoryItem(String d)
27     {
28         description = d;
29         units = 0;
30     }
31
32     /**
33     * The following constructor accepts a
34     * String argument that is assigned to the
35     * description field, and an int argument
36     * that is assigned to the units field.
37     */
38
39     public InventoryItem(String d, int u)
40     {
41         description = d;
42         units = u;
43     }
44
45     /**
46     * The setDescription method assigns its
47     * argument to the description field.
48     */
49
50     public void setDescription(String d)
51     {
52         description = d;
53     }
54
55     /**
56     * The setUnits method assigns its argument
57     * to the units field.
58     */
59
60     public void setUnits(int u)
61     {
62         units = u;
63     }
64
65     /**
66     * The getDescription method returns the
67     * value in the description field.
68     */
```

```

69     public String getDescription()
70     {
71         return description;
72     }
73
74
75     /**
76      * The getUnits method returns the value in
77      * the units field.
78      */
79
80     public int getUnits()
81     {
82         return units;
83     }
84 }
```

The first constructor in the `InventoryItem` class (in lines 14 through 18) is the no-arg constructor. It assigns an empty string ("") to the `description` field, and assigns 0 to the `units` field. The second constructor, in lines 26 through 30, accepts a `String` argument, `d`, which is assigned to the `description` field, and assigns 0 to the `units` field. The third constructor, in lines 39 through 43, accepts a `String` argument, `d`, which is assigned to the `description` field, and an `int` argument, `u`, which is assigned to the `units` field. The program in [Code Listing 6-12](#) demonstrates the class.

Code Listing 6-12 (`InventoryDemo.java`)

```

1  /**
2   * This program demonstrates the InventoryItem class's
3   * three constructors.
4   */
5
6  public class InventoryDemo
7  {
8      public static void main(String[] args)
9      {
10          // Variables to reference 3 instances of
11          // the InventoryItem class.
12          InventoryItem item1, item2, item3;
```

```

13
14     // Instantiate item1 and use the
15     // no-arg constructor.
16     item1 = new InventoryItem();
17     System.out.println("Item 1:");
18     System.out.println("Description: " +
19                     item1.getDescription());
20     System.out.println("Units: " + item1.getUnits());
21     System.out.println();
22
23
24     // Instantiate item2 and use the
25     // second constructor.
26     item2 = new InventoryItem("Wrench");
27     System.out.println("Item 2:");
28     System.out.println("Description: " +
29                     item2.getDescription());
30     System.out.println("Units: " + item2.getUnits());
31     System.out.println();
32
33     // Instantiate item3 and use the
34     // third constructor.
35     item3 = new InventoryItem("Hammer", 20);
36     System.out.println("Item 3:");
37     System.out.println("Description: " +
38                     item3.getDescription());
39     System.out.println("Units: " + item3.getUnits());
40 }
41 }
```

Program Output

Item 1: Description: Units: 0 Item 2: Description: Wrench Units: 0 Item 3: Description: Hammer Units: 20



Checkpoint

1. 6.4 Is it required that overloaded methods have different return values, different parameter lists, or both?
2. 6.5 What is a method's signature?
3. 6.6 What will the following program display?

```
public class CheckPoint
{
    public static void main(String[] args)
    {
        message(1.2);
        message(1);
    }
    public static void message(int x)
    {
        System.out.print("This is the first version ");
        System.out.println("of the method.");
    }
    public static void message(double x)
    {
        System.out.print("This is the second version ");
        System.out.println("of the method.");
    }
}
```

4. 6.7 How many default constructors may a class have?

6.4 Passing Objects as Arguments to Methods

Concept:

To pass an object as a method argument, you pass an object reference.

In [Chapter 3](#), we discussed how primitive variables can be passed as arguments to methods. You can also pass objects as arguments to methods. For example, look at [Code Listing 6-13](#).

Code Listing 6-13 (PassObject.java)

```
1  /**
2   * This program passes an object as an argument.
3   */
4
5 public class PassObject
6 {
7     public static void main(String[] args)
8     {
9         // Create an InventoryItem object.
10        InventoryItem item = new InventoryItem("Wrench", 20);
11
12        // Pass the object to the DisplayItem method.
13        displayItem(item);
14    }
15
16 /**
17  * The following method accepts an InventoryItem
18  * object as an argument and displays its contents.
19  */
```

```
20
21     public static void displayItem(InventoryItem i)
22     {
23         System.out.println("Description: " + i.getDescription())
24         System.out.println("Units: " + i.getUnits());
25     }
26 }
```

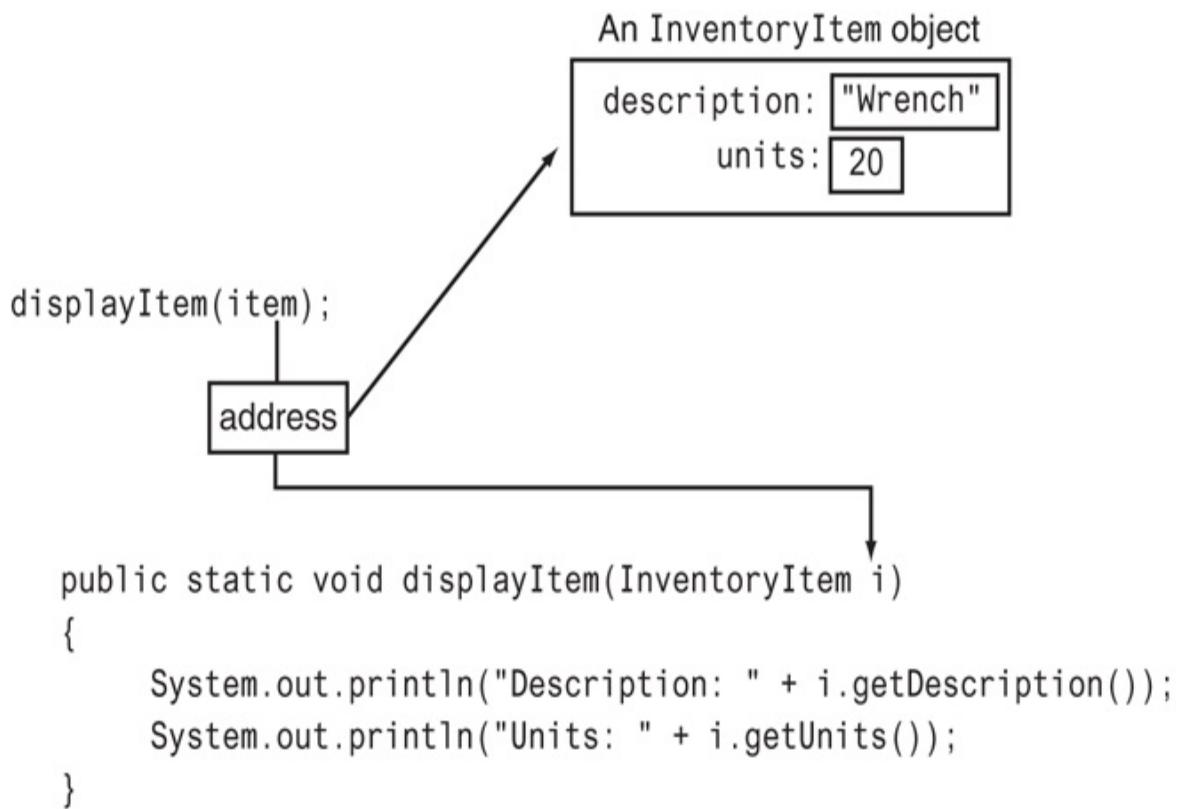
Program Output

```
Description: Wrench
Units: 20
```

When an object is passed as an argument, it is actually a reference to the object that is passed. In this program's `main` method, the `item` variable is an `InventoryItem` reference variable. Its value is passed as an argument to the `displayItem` method. The `displayItem` method has a parameter variable, `i`, also an `InventoryItem` reference variable, which receives the argument.

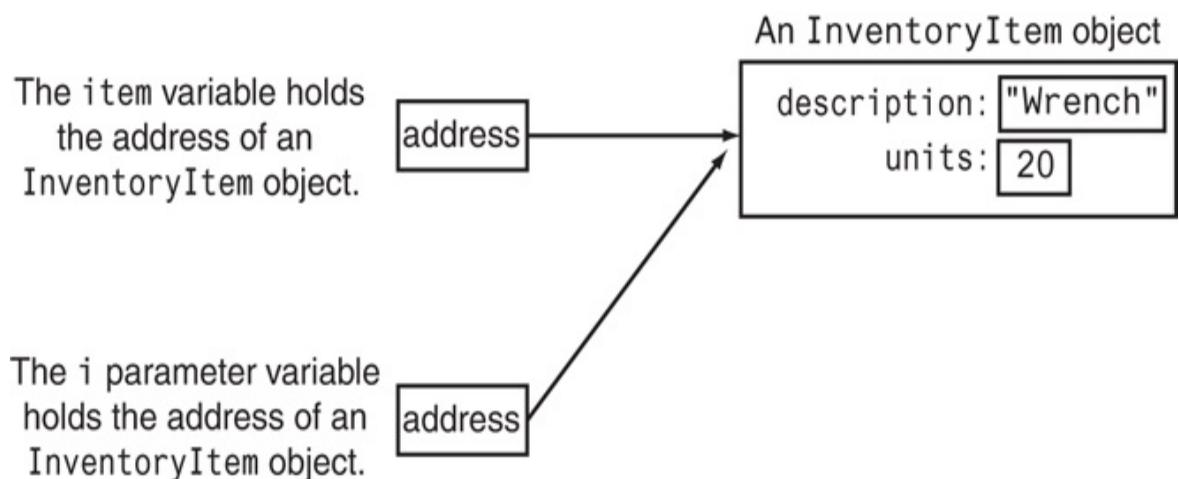
Recall that a reference variable holds the memory address of an object. When the `displayItem` method is called, the address that is stored in `item` is passed into the `i` parameter variable. This is illustrated in [Figure 6-3](#). This means that when the `displayItem` method is executing, `item` and `i` both reference the same object. This is illustrated in [Figure 6-4](#).

Figure 6-3 Passing a reference as an argument



[Figure 6-3 Full Alternative Text](#)

Figure 6-4 Both item and i reference the same object



[Figure 6-4 Full Alternative Text](#)

Recall from [Chapter 3](#) that when a variable is passed as an argument to a method, it is said to be *passed by value*. This means that a copy of the variable's value is passed into the method's parameter. When the method changes the contents of the parameter variable, it does not affect the contents of the actual variable that was passed as an argument. When a reference variable is passed as an argument to a method, however, the method has access to the object that the variable references. As you can see from [Figure 6-4](#), the `displayItem` method has access to the same `InventoryItem` object that the `item` variable references. When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable. This is demonstrated in [Code Listing 6-14](#).

Code Listing 6-14 (`PassObject2.java`)

```
1  /**
2   * This program passes an object as an argument.
3   * The object is modified by the receiving method.
4   */
5
6  public class PassObject2
7  {
8      public static void main(String [] args)
9      {
10         // Create an InventoryItem object.
11         InventoryItem item = new InventoryItem("Wrench", 20);
12
13         // Display the object's contents.
14         System.out.println("The contents of item are:");
15         System.out.println("Description: " +
16                           item.getDescription() +
17                           " Units: " + item.getUnits());
18
19         // Pass the object to the ChangeItem method.
20         changeItem(item);
21
22         // Display the object's contents again.
```

```

23     System.out.println();
24     System.out.println("Now the contents of item are:");
25     System.out.println("Description: " +
26                         item.getDescription() +
27                         " Units: " + item.getUnits());
28 }
29 /**
30 * The following method accepts an InventoryItem
31 * object as an argument and changes its contents.
32 */
33
34
35     public static void changeItem(InventoryItem i)
36     {
37         i.setDescription("Hammer");
38         i.setUnits(5);
39     }
40 }
```

Program Output

```

The contents of item are:
Description: Wrench  Units: 20

Now the contents of item are:
Description: Hammer  Units: 5
```

When writing a method that receives a reference as an argument, you must take care not to accidentally modify the contents of the object that is passed.

In the Spotlight: Simulating the Game of Cho–Han



Cho–Han is a traditional Japanese gambling game in which a dealer uses a cup to roll two six-sided dice. The cup is placed upside down on a table so that the value of the dice is concealed. Players then wager on whether the sum of the dice values is even (Cho) or odd (Han). The winner, or winners,

take all of the wagers, or the house takes them if there are no winners.

We will develop a program that simulates a simplified variation of the game. The simulated game will have a dealer and two players. The players will not wager money, but will simply guess whether the sum of the dice values is even (Cho) or odd (Han). One point will be awarded to the player, or players, correctly guessing the outcome. The game is played for five rounds, and the player with the most points is the grand winner.

In the program, we will use the `Die` class that was introduced in [Chapter 4](#). We will create two instances of the class to represent two six-sided dice. In addition to the `Die` class, we will write the following classes:

- `Dealer` class: We will create an instance of this class to represent the dealer. It will have the ability to roll the dice, report the value of the dice, and report whether the total dice value is Cho or Han.
- `Player` class: We will create two instances of this class to represent the players. Instances of the `Player` class can store the player's name, make a guess between Cho or Han, and be awarded points.

First, let's look at the `Dealer` class. [Figure 6-5](#) shows a UML diagram for the class, and [Code Listing 6-15](#) shows the code.

Figure 6-5 UML diagram for the Dealer class

Dealer
- die1Value : int - die2Value : int
+ Dealer() + rollDice() : void + getChoOrHan() : String + getDie1Value() : int + getDie2Value() : int

[Figure 6-5 Full Alternative Text](#)

Code Listing 6-15 (Dealer.java)

```

1  /**
2   * Dealer class for the game of Cho-Han
3   */
4
5  public class Dealer
6  {
7      private int die1Value; // The value of die #1
8      private int die2Value; // The value of die #2
9
10     /**
11      * Constructor
12     */
13
14     public Dealer()
15     {
16         die1Value = 0;
17         die2Value = 0;
18     }
19
20     /**
21      * The rollDice method rolls the dice and saves
22      * their values.
23     */
24
25     public void rollDice()
26     {

```

```
27     final int SIDES = 6; // Number of sides for the dice
28
29     // Create the two dice. (This also rolls them.)
30     Die die1 = new Die(SIDES);
31     Die die2 = new Die(SIDES);
32
33     // Record their values.
34     die1Value = die1.getValue();
35     die2Value = die2.getValue();
36 }
37
38 /**
39 *   The getChoOrHan method returns the result of the dice
40 *   roll. If the sum of the dice is even, the method return
41 *   "Cho (even)". Otherwise, it returns "Han (odd)".
42 */
43
44 public String getChoOrHan()
45 {
46     String result; // To hold the result
47
48     // Get the sum of the dice.
49     int sum = die1Value + die2Value;
50
51     // Determine even or odd.
52     if (sum % 2 == 0)
53         result = "Cho (even)";
54     else
55         result = "Han (odd)";
56
57     // Return the result.
58     return result;
59 }
60
61 /**
62 *   The getDie1Value method returns the value of
63 *   die #1.
64 */
65
66 public int getDie1Value()
67 {
68     return die1Value;
69 }
70
71 /**
72 *   The getDie2Value method returns the value of
73 *   die #2.
74 */
```

```
75     public int getDie2Value()
76     {
77         return die2Value;
78     }
79 }
```

Let's take a closer look at the code for the `Dealer` class:

- Lines 7 and 8 declare the fields `die1Value` and `die2Value`. These fields will hold the value of the two dice after they have been rolled.
- The constructor, in lines 14 through 18, initializes the `die1Value` and `die2Value` fields to 0.
- The `rollDice` method, in lines 25 through 36, simulates the rolling of the dice. Lines 30 and 31 create two `Die` objects. Recall that the `Die` class constructor performs an initial roll of the die, so there is no need to call the `Die` objects' `roll` method. Lines 34 and 35 save the value of the dice in the `die1Value` and `die2Value` fields.
- The `getChoOrHan` method, in lines 44 through 59, returns a string indicating whether the sum of the dice is Cho (even) or Han (odd).
- The `getDie1Value` method, in lines 66 through 69, returns the value of first die (stored in the `die1Value` field).
- The `getDie2Value` method, in lines 76 through 79, returns the value of second die (stored in the `die2Value` field).

Now let's look at the `Player` class. [Figure 6-6](#) shows a UML diagram for the class, and [Code Listing 6-16](#) shows the code.

Figure 6-6 UML diagram for the Player class

Player
<ul style="list-style-type: none"> - name : String - guess : String - points : int
<ul style="list-style-type: none"> + Player(playerName : String) + makeGuess() : void + addPoints(newPoints : int) : void + getName() : String + getGuess() : String + getPoints() : int

[Figure 6-6 Full Alternative Text](#)

Code Listing 6-16 (Player.java)

```

1 import java.util.Random;
2
3 /**
4  * Player class for the game of Cho-Han
5 */
6
7 public class Player
8 {
9     private String name;      // The player's name
10    private String guess;    // The player's guess
11    private int points;      // The player's points
12
13    /**
14     * Constructor
15     * Accepts the player's name as an argument
16     */
17
18    public Player(String playerName)
19    {
20        name = playerName;
21        guess = "";
22        points = 0;
23    }
24

```

```
25  /**
26   * The makeGuess method causes the player to guess
27   * either "Cho (even)" or "Han (odd)".
28   */
29
30  public void makeGuess()
31  {
32      // Create a Random object.
33      Random rand = new Random();
34
35      // Get a random number, either 0 or 1.
36      int guessNumber = rand.nextInt(2);
37
38      // Convert the random number to a guess of
39      // either "Cho (even)" or "Han (odd)".
40      if (guessNumber == 0)
41          guess = "Cho (even)";
42      else
43          guess = "Han (odd)";
44  }
45
46  /**
47   * The addPoints method adds a specified number of
48   * points to the player's current balance. The number
49   * of points is passed as an argument.
50   */
51
52  public void addPoints(int newPoints)
53  {
54      points += newPoints;
55  }
56
57  /**
58   * The getName method returns the player's name.
59   */
60
61  public String getName()
62  {
63      return name;
64  }
65
66  /**
67   * The getGuess method returns the player's guess.
68   */
69
70  public String getGuess()
71  {
72      return guess;
```

```
73     }
74
75     /**
76      The getPoints method returns the player's points
77     */
78
79     public int getPoints()
80     {
81         return points;
82     }
83 }
```

Here's a summary of the code for the `Player` class:

- Lines 9 through 11 declare the fields `name`, `guess`, and `points`. These fields will hold the player's name, the player's guess, and the number of points the player has earned.
- The constructor, in lines 18 through 23, accepts an argument for the player's name, which is assigned to the `name` field. The `guess` field is assigned an empty string, and the `points` field is set to 0.
- The `makeGuess` method, in lines 30 through 44, causes the player to make a guess. The method generates a random number that is either a 0 or a 1. The `if` statement that begins at line 40 assigns the string "Cho (even)" to the `guess` field if the random number is 0, or it assigns the string "Han (odd)" to the `guess` field if the random number is 1.
- The `addPoints` method, in lines 52 through 55, adds the number of points specified by the argument to the player's `point` field.
- The `getName` method, in lines 61 through 64, returns the player's name.
- The `getGuess` method, in lines 70 through 73, returns the player's `guess`.
- The `getPoints` method, in lines 79 through 82, returns the player's `points`.

[Code Listing 6-17](#) shows the program that uses these classes to simulate the game. The `main` method simulates five rounds of the game, displaying the results of each round, then displays the overall game results.

Code Listing 6-17 (ChoHan.java)

```
1 import java.util.Scanner;
2
3 public class ChoHan
4 {
5     public static void main(String[] args)
6     {
7         final int MAX_ROUNDS = 5;    // Number of rounds
8         String player1Name;        // First player's name
9         String player2Name;        // Second player's name
10
11        // Create a Scanner object for keyboard input.
12        Scanner keyboard = new Scanner(System.in);
13
14        // Get the player's names.
15        System.out.print("Enter the first player's name: ");
16        player1Name = keyboard.nextLine();
17        System.out.print("Enter the second player's name: ");
18        player2Name = keyboard.nextLine();
19
20        // Create the dealer.
21        Dealer dealer = new Dealer();
22
23        // Create the two players.
24        Player player1 = new Player(player1Name);
25        Player player2 = new Player(player2Name);
26
27        // Play the rounds.
28        for (int round = 0; round < MAX_ROUNDS; round++)
29        {
30            System.out.println("-----");
31            System.out.printf("Now playing round %d.\n", round
32
33            // Roll the dice.
34            dealer.rollDice();
35
36            // The players make their guesses.
37            player1.makeGuess();
38            player2.makeGuess();
39
40            // Determine the winner of this round.
41            roundResults(dealer, player1, player2);
42        }
43
```

```
44     // Display the grand winner.
45     displayGrandWinner(player1, player2);
46 }
47 /**
48 * The roundResults method determines the results of
49 * the current round. The parameters are:
50 * dealer: The Dealer object
51 * player1: Player #1 object
52 * player2: Player #2 object
53 */
54
55 public static void roundResults(Dealer dealer, Player pla
56                                 Player player2)
57 {
58     // Show the dice values.
59     System.out.printf("The dealer rolled %d and %d.\n",
60                       dealer.getDie1Value(), dealer.getDie
61                       System.out.printf("Result: %s\n", dealer.getChoOrHan()
62
63     // Check each player's guess and award points.
64     checkGuess(player1, dealer);
65     checkGuess(player2, dealer);
66 }
67
68 /**
69 * The checkGuess method checks a player's guess against
70 * the dealer's result. The parameters are:
71 * player: The Player object to check.
72 * dealer: The Dealer object.
73 */
74
75 public static void checkGuess(Player player, Dealer deale
76 {
77     final int POINTS_TO_ADD = 1; // Points to award winner
78     String guess = player.getGuess(); // Player'
79     String choHanResult = dealer.getChoOrHan(); // Cho or
80
81     // Display the player's guess.
82     System.out.printf("The player %s guessed %s.\n",
83                       player.getName(), player.getGuess())
84
85     // Award points if the player guessed correctly.
86     if (guess.equalsIgnoreCase(choHanResult))
87     {
88         player.addPoints(POINTS_TO_ADD);
89         System.out.printf("Awarding %d point(s) to %s.\n",
90                           POINTS_TO_ADD, player.getName());
91     }
92 }
```

```

92         }
93     }
94
95     /**
96      * The displayGrandWinner method displays the game's gra
97      * The parameters are:
98      * player1: Player #1
99      * player2: Player #2
100     */
101
102    public static void displayGrandWinner(Player player1, Pla
103    {
104        System.out.println("-----");
105        System.out.println("Game over. Here are the results:");
106        System.out.printf("%s: %d points.\n", player1.getName(
107                                player1.getPoints()));
108        System.out.printf("%s: %d points.\n", player2.getName(
109                                player2.getPoints());
110
111        if (player1.getPoints() > player2.getPoints())
112            System.out.println(player1.getName() + " is the gra
113        else if (player2.getPoints() > player1.getPoints())
114            System.out.println(player2.getName() + " is the gra
115        else
116            System.out.println("Both players are tied!");
117    }
118 }

```

Program Output with Example Input Shown in Bold

Enter the first player's name: **Chelsea** 

Enter the second player's name: **Chris** 

Now playing round 1.
The dealer rolled 3 and 6.
Result: Han (odd)
The player Chelsea guessed Han (odd).
Awarding 1 point(s) to Chelsea.
The player Chris guessed Han (odd).
Awarding 1 point(s) to Chris.

Now playing round 2.
The dealer rolled 4 and 5.
Result: Han (odd)
The player Chelsea guessed Cho (even).
The player Chris guessed Cho (even).

```
Now playing round 3.  
The dealer rolled 5 and 6.  
Result: Han (odd)  
The player Chelsea guessed Cho (even).  
The player Chris guessed Han (odd).  
Awarding 1 point(s) to Chris.
```

```
Now playing round 4.  
The dealer rolled 1 and 6.  
Result: Han (odd)  
The player Chelsea guessed Cho (even).  
The player Chris guessed Cho (even).
```

```
Now playing round 5.  
The dealer rolled 6 and 6.  
Result: Cho (even)  
The player Chelsea guessed Han (odd).  
The player Chris guessed Cho (even).  
Awarding 1 point(s) to Chris.
```

Game over. Here are the results:

Chelsea: 1 points.
Chris: 3 points.
Chris is the grand winner!

Let's look at the code. Here is a summary of the `main` method:

- Lines 7 through 9 make the following declarations: `MAX_ROUNDS` (the number of rounds to play) `player1Name` (to hold the name of player #1), and `player2Name` (to hold the name of player #2).
- Lines 15 through 18 prompt the user to enter the players' names.
- Line 21 creates an instance of the `Dealer` class. The object represents the dealer and is referenced by the `dealer` variable.
- Line 24 creates an instance of the `Player` class. The object represents player #1 and is referenced by the `player1` variable. Notice that `player1Name` is passed as an argument to the constructor.
- Line 25 creates another instance of the `Player` class. The object represents player #2 and is referenced by the `player2` variable. Notice that `player2Name` is passed as an argument to the constructor.

- The `for` loop that begins in line 28 iterates five times, causing the simulation of five rounds of the game. The loop performs the following actions:
 - Line 34 causes the dealer to roll the dice.
 - Line 37 causes player #1 to make a guess (Cho or Han).
 - Line 38 causes player #2 to make a guess (Cho or Han).
- Line 41 passes the `dealer`, `player1`, and `player2` objects to the `roundResults` method. The method displays the results of this round.
- Line 45 passes the `player1` and `player2` objects to the `displayGrandWinner` method, which displays the grand winner of the game.

The `roundResults` method, which displays the results of a round, appears in lines 56 through 67. Here is a summary of the method:

- The method accepts references to the `dealer`, `player1`, and `player2` objects as arguments.
- The statement in lines 60 and 61 displays the value of the two dice.
- Line 62 calls the `dealer` object's `getChoOrHan` method to display the results, Cho or Han.
- Line 65 calls the `checkGuess` method, passing the `player1` and `dealer` objects as arguments. The `checkGuess` method compares a player's guess to the dealer's result (Cho or Han), and awards points to the player if the guess is correct.
- Line 66 calls the `checkGuess` method, passing the `player2` and `dealer` objects as arguments.

The `checkGuess` method, which compares a player's guess to the dealer's result, awarding points

to the player for a correct guess, appears in lines 76 through 93. Here is a summary of the method:

- The method accepts references to a `Player` object and the `Dealer` object as arguments.
- Line 78 declares the constant `POINTS_TO_ADD`, set to the value 1, which is the number of points to add to the player's balance if the player's guess is correct.
- Line 79 assigns the player's guess to the `String` object `guess`.
- Line 80 assigns the dealer's results (Cho or Han) to the `String` object `choHanResult`.
- The statement in lines 83 and 84 displays the player's name and guess.
- The `if` statement in line 87 compares the player's guess to the dealer's result. If they match, then the player guessed correctly, and line 89 awards points to the player.

The `displayGrandWinner` method, which displays the grand winner of the game, appears in lines 102 through 117. Here is a summary of the method:

- The method accepts references to the `player1` and `player2` objects.
- The statements in lines 106 through 109 display both players' names and points.
- The `if-else-if` statement that begins in line 111 determines which of the two players has the highest score, and displays that player's name as the grand winner. If both players have the same score, a tie is declared.



Checkpoint

1. 6.8 When an object is passed as an argument to a method, what is actually passed?

2. 6.9 When an argument is passed by value, the method has a copy of the argument and does not have access to the original argument. Is this still true when an object is passed to a method?
3. 6.10 Recall the `Rectangle` class shown earlier in this chapter. Write a method that accepts a `Rectangle` object as its argument and displays the object's `length` and `width` fields on the screen.

6.5 Returning Objects from Methods

Concept:

A method can return a reference to an object.



VideoNote **Returning Objects from Methods**

Just as methods can be written to return an `int`, `double`, `float`, or other primitive data type, they can also be written to return a reference to an object. For example, the program in [Code Listing 6-18](#) uses a method, `getData`, which returns a reference to an `InventoryItem` object.

Code Listing 6-18 (`ReturnObject.java`)

```
1 import java.util.Scanner;
2
3 /**
4  * This program demonstrates how a method can return
5  * a reference to an object.
6 */
7
8 public class ReturnObject
9 {
10     public static void main(String[] args)
11     {
12         // Declare a variable that will be used to
13         // reference an InventoryItem object.
```

```

14     InventoryItem item;
15
16     // The getData method will return a reference
17     // to an InventoryItem object.
18
19     item = getData();
20
21     // Display the object's data.
22     System.out.println("Here is the data you entered:");
23     System.out.println("Description: " +
24                     item.getDescription() +
25                     " Units: " + item.getUnits());
26
27 }
28
29 /**
30 * The getData method gets an item's description
31 * and number of units from the user. The method
32 * returns an InventoryItem object containing
33 * the data that was entered.
34 */
35
36 public static InventoryItem getData()
37 {
38     String desc;    // To hold the description
39     int units;      // To hold the units
40
41     // Create a Scanner object for keyboard input.
42     Scanner keyboard = new Scanner(System.in);
43
44     // Get the item description.
45     System.out.print("Enter an item description: ");
46     desc = keyboard.nextLine();
47
48     // Get the number of units.
49     System.out.print("Enter a number of units: ");
50     units = keyboard.nextInt();
51
52     // Create an InventoryItem object and return
53     // a reference to it.
54     return new InventoryItem(desc, units);
55 }
56 }
```

Program Output with Example Input Shown in Bold

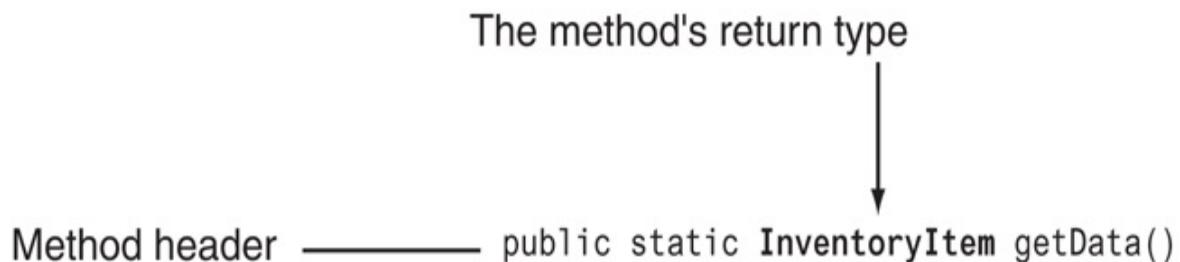
Enter an item description: **Pliers** 

```
Enter a number of units: 25 
Here is the data you entered:
Description: Pliers Units: 25
```

Notice in line 36 the `getData` method has the return data type of `InventoryItem`.

[Figure 6-7](#) shows the method's return type, which is listed in the method header.

Figure 6-7 The `getData` method header



[Figure 6-7 Full Alternative Text](#)

A return type of `InventoryItem` means the method returns a reference to an `InventoryItem` object when it terminates. The following statement, which appears in line 19 in the `main` method, assigns the `getData` method's return value to `item`:

```
item = getData();
```

After this statement executes, the `item` variable will reference the `InventoryItem` object that was returned from the `getData` method.

Now let's look at the `getData` method. First, the method declares two local variables, `desc` and `units`. These variables are used to hold an item description and a number of units, as entered by the user in lines 46 and 50. The last statement in the `getData` method is the following `return` statement,

which appears in line 54:

```
return new InventoryItem(desc, units);
```

This statement uses the `new` key word to create an `InventoryItem` object, passing `desc` and `units` as arguments to the constructor. The address of the object is then returned from the method.



Checkpoint

1. 6.11 Recall the `Rectangle` class shown earlier in this chapter. Write a method that returns a reference to a `Rectangle` object. The method should store the user's input in the object's `length` and `width` fields before returning it.

6.6 The `toString` Method

Concept:

Most classes can benefit from having a method named `toString`, which is implicitly called under certain circumstances. Typically, the method returns a string that represents the state of an object.

So far, you've seen many examples in which an object is created then its contents are used in messages displayed on the screen. Previously, you saw the following statement in lines 23 through 25 of [Code Listing 6-18](#):

```
System.out.println("Description: " +
    item.getDescription() +
    " Units: " + item.getUnits());
```

Recall that `item` references an `InventoryItem` object. In this statement, the `System.out .println` method displays a string showing the values of the object's `description` and `units` fields. Assuming that the object's `description` field is set to "Pliers" and the `units` field is set to 25, the output of this statement will look like this:

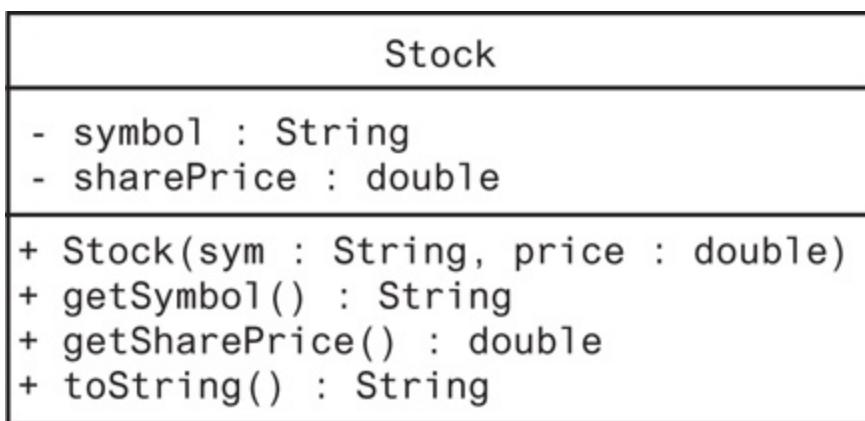
```
Description: Pliers  Units: 25
```

In this statement, the argument passed to `System.out.println` is a string, which is put together from several pieces. The concatenation operator (+) joins the pieces together. The first piece is the string literal "Description: ". To this, the value returned from the `getDescription` method is concatenated, followed by the string literal " Units: ", followed by the value returned from the `getUnits` method. The resulting string represents the current state of the object.

Creating a string that represents the state of an object is such a common task

that many programmers equip their classes with a method that returns such a string. In Java, it is standard practice to name this method `toString`. Let's look at an example of a class that has a `toString` method. [Figure 6-8](#) shows the UML diagram for the `Stock` class, which holds data about a company's stock.

Figure 6-8 UML diagram for the Stock class



[Figure 6-8 Full Alternative Text](#)

This class has two fields: `symbol` and `sharePrice`. The `symbol` field holds the trading `symbol` for the company's stock. This is a short series of characters used to identify the stock on the stock exchange. For example, the XYZ Company's stock might have the trading symbol XYZ. The `sharePrice` field holds the current price per share of the stock. [Table 6-1](#) describes the class's methods.

Table 6-1 The Stock class methods

Method	Description
--------	-------------

Constructor	This constructor accepts arguments that are assigned to the <code>symbol</code> and <code>sharePrice</code> fields.
<code>getSymbol</code>	This method returns the value in the <code>symbol</code> field.
<code>getSharePrice</code>	This method returns the value in the <code>sharePrice</code> field.
<code>toString</code>	This method returns a string representing the state of the object. The string will be appropriate for displaying on the screen.

[Code Listing 6-19](#) shows the code for the `Stock` class. (If you have downloaded the book's source code, you will find this file in the folder [*Chapter 06\Stock Class Phase 1.*](#))

Code Listing 6-19 (Stock.java)

```

1  /**
2   * The Stock class holds data about a stock.
3   */
4
5  public class Stock
6  {
7      private String symbol;      // Trading symbol of stock
8      private double sharePrice; // Current price per share
9
10     /**
11      * The constructor accepts arguments for the
12      * stock's trading symbol and share price.
13     */
14
15     public Stock(String sym, double price)
16     {
17         symbol = sym;
18         sharePrice = price;
19     }
20
21     /**
22      * getSymbol method
23     */
24
25     public String getSymbol()
26     {

```

```
27         return symbol;
28     }
29
30     /**
31      * getSharePrice method
32      */
33
34     public double getSharePrice()
35     {
36         return sharePrice;
37     }
38
39     /**
40      * toString method
41      */
42
43     public String toString()
44     {
45         // Create a string describing the stock.
46         String str = "Trading symbol: " + symbol +
47                         "\nShare price: " + sharePrice;
48
49         // Return the string.
50         return str;
51     }
52 }
```

The `toString` method appears in lines 43 through 51. The method creates a string listing the stock's trading symbol and price per share. This string is then returned from the method. A call to the method can then be passed to `System.out.println`, as shown in the following code.

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany.toString());
```

This code would produce the following output:

```
Trading symbol: XYZ
Share price: 9.62
```

In actuality, it is unnecessary to explicitly call the `toString` method in this example. If you write a `toString` method for a class, Java will automatically call the method when the object is passed as an argument to `print` or `println`. The following code would produce the same output as that

previously shown:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println(xyzCompany);
```

Java also implicitly calls an object's `toString` method any time you concatenate an object of the class with a string. For example, the following code would implicitly call the `xyzCompany` object's `toString` method:

```
Stock xyzCompany = new Stock ("XYZ", 9.62);
System.out.println("The stock data is:\n" + xyzCompany);
```

This code would produce the following output:

```
The stock data is:
Trading symbol: XYZ
Share price: 9.62
```

[Code Listing 6-20](#) shows a complete program demonstrating the `Stock` class's `toString` method. (If you have downloaded the book's source code, you will find this file in the folder [Chapter 06\Stock Class Phase 1.](#))

Code Listing 6-20 (StockDemo1.java)

```
1 /**
2  * This program demonstrates the Stock class's
3  * toString method.
4 */
5
6 public class StockDemo1
7 {
8     public static void main(String[] args)
9     {
10         // Create a Stock object for the XYZ Company.
11         // The trading symbol is XYZ and the current
12         // price per share is $9.62.
13         Stock xyzCompany = new Stock ("XYZ", 9.62);
14
15         // Display the object's values.
16         System.out.println(xyzCompany);
```

```
17 }  
18 }
```

Program Output

```
Trading symbol: XYZ  
Share price: 9.62
```



Note:

Every class automatically has a `toString` method that returns a string containing the object's class name, followed by the @ symbol, followed by an integer unique to the object. This method is called when necessary, if you have not provided your own `toString` method. You will learn more about this in [Chapter 9](#).

6.7 Writing an equals Method

Concept:

You cannot determine whether two objects contain the same data by comparing them with the == operator. Instead, the class must have a method such as equals for comparing the contents of objects.

Recall from [Chapter 4](#) that the String class has a method named equals that determines whether two strings are equal. You can write an equals method for any of your own classes as well.

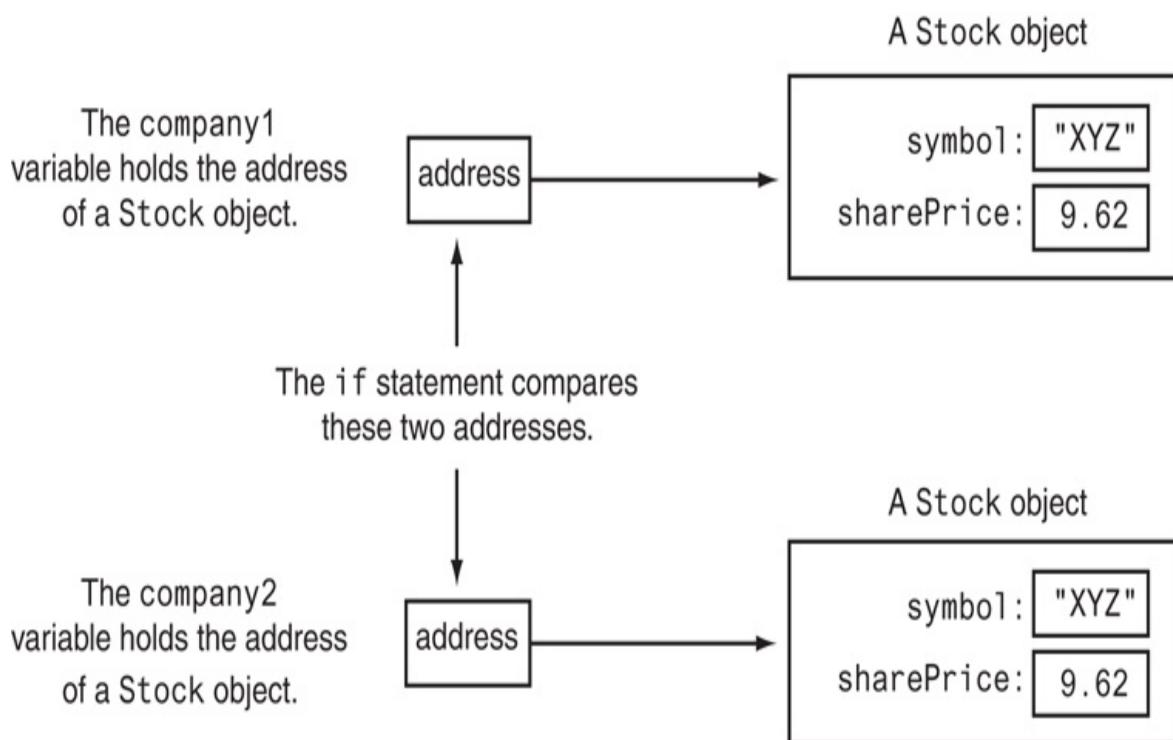
In fact, you must write an equals method (or one that works like it) for a class in order to determine whether two objects of the class contain the same values. This is because you cannot use the == operator to compare the contents of two objects. For example, the following code might appear to compare the contents of two Stock objects, but in reality it does not.

```
// Create two Stock objects with the same values.  
Stock company1 = new Stock("XYZ", 9.62);  
Stock company2 = new Stock("XYZ", 9.62);  
// Use the == operator to compare the objects.  
// (This is a mistake.)  
if (company1 == company2)  
    System.out.println("Both objects are the same.");  
else  
    System.out.println("The objects are different.");
```

When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. This is illustrated in [Figure 6-9](#).

Figure 6-9 The if statement

tests the contents of the reference variables, not the contents of the objects that the variables reference



[Figure 6-9 Full Alternative Text](#)

Because the two variables reference different objects in memory, they will contain different addresses. Therefore, the result of the boolean expression `company1 == company2` is `false`, and the code reports that the objects are not the same. Instead of using the `==` operator to compare the two `Stock` objects, we should write an `equals` method that compares the contents of the two objects.

If you have downloaded the book's source code, in the folder [Chapter 06\Stock Class Phase 2](#), you will find a revision of the `Stock` class. This

version of the class has an `equals` method. The code for the method follows. (No other part of the class has changed, so

only the `equals` method is shown.)

```
public boolean equals(Stock object2)
{
    boolean status;

    // Determine whether this object's symbol and
    // sharePrice fields are equal to object2's
    // symbol and sharePrice fields.
    if (symbol.equals(object2.symbol) && sharePrice == object2.sha
        status = true; // Yes, the objects are equal.
    else
        status = false; // No, the objects are not equal.

    // Return the value in status.
    return status;
}
```

The `equals` method accepts a `Stock` object as its argument. The parameter variable `object2` will reference the object that was passed as an argument. The `if` statement performs the following comparison: If the `symbol` field of the calling object is equal to the `symbol` field of `object2`, and the `sharePrice` field of the calling object is equal to the `sharePrice` field of `object2`, then the two objects contain the same values. In this case, the local variable `status` (a `boolean`) is set to `true`. Otherwise, `status` is set to `false`. Finally, the method returns the value of the `status` variable.

Notice that the method can access `object2`'s `symbol` and `sharePrice` fields directly. Because `object2` references a `Stock` object, and the `equals` method is a member of the `Stock` class, the method is allowed to access `object2`'s private fields.

The program in [Code Listing 6-21](#) demonstrates the `equals` method. (If you have downloaded the book's source code, you will find this file in the folder [*Chapter 06\Stock Class Phase 2.*](#))

Code Listing 6-21

(StockCompare.java)

```
1  /**
2   * This program uses the Stock class's equals
3   * method to compare two Stock objects.
4   */
5
6  public class StockCompare
7  {
8      public static void main(String[] args)
9      {
10         // Create two Stock objects with the same values.
11         Stock company1 = new Stock("XYZ", 9.62);
12         Stock company2 = new Stock("XYZ", 9.62);
13
14         // Use the equals method to compare the objects.
15         if (company1.equals(company2))
16             System.out.println("Both objects are the same.");
17         else
18             System.out.println("The objects are different.");
19     }
20 }
```

Program Output

Both objects are the same.

If you want to be able to compare the objects of a given class, you should always write an equals method for the class.



Note:

Every class automatically has an equals method that works the same as the == operator. This method is called when necessary if you have not provided your own equals method. You will learn more about this in [Chapter 9](#).

6.8 Methods That Copy Objects

Concept:

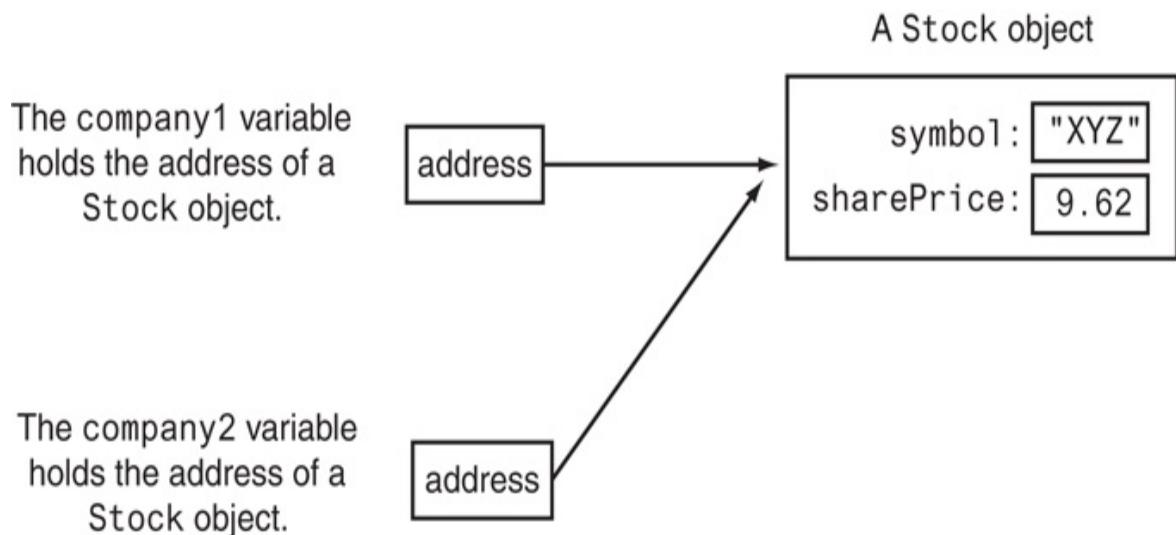
You can simplify the process of duplicating objects by equipping a class with a method that returns a copy of an object.

You cannot make a copy of an object with a simple assignment statement as you would with a primitive variable. For example, look at the following code:

```
Stock company1 = new Stock("XYZ", 9.62);  
Stock company2 = company1;
```

The first statement creates a Stock object and assigns its address to the company1 variable. The second statement assigns company1 to company2. This does not make a copy of the object referenced by company1. Rather, it makes a copy of the address that is stored in company1, and stores that address in company2. After this statement executes, both the company1 and company2 variables will reference the same object. This is illustrated in [Figure 6-10](#).

**Figure 6-10 Both variables
reference the same object**



[Figure 6-10 Full Alternative Text](#)

This type of assignment operation is called a *reference copy* because only the object's address is copied, not the actual object itself. To copy the object itself, you must create a new object, then set the new object's fields to the same values as the fields of the object being copied. This process can be simplified by equipping the class with a method that performs this operation. The method then returns a reference to the duplicate object.

If you have downloaded the book's source code, in the folder [Chapter 06\Stock Class Phase 3](#) you will find a revision of the Stock class. This version of the class has a method named `copy` that returns a copy of a Stock object. The code for the method follows. (No other part of the class has changed, so only the `copy` method is shown.)

```
public Stock copy()
{
    // Create a new Stock object and initialize it
    // with the same data held by the calling object.
    Stock copyObject = new Stock(symbol, sharePrice);

    // Return a reference to the new object.
    return copyObject;
}
```

The `copy` method creates a new Stock object and passes the calling object's symbol

and sharePrice fields as arguments to the constructor. This makes the new object a

copy of the calling object. The program in [Code Listing 6-22](#) demonstrates the copy method. (If you have downloaded the book's source code, you will find this file in the folder [Chapter 06\Stock Class Phase 3.](#))

Code Listing 6-22 (ObjectCopy.java)

```
1  /**
2   * This program uses the Stock class's copy method
3   * to create a copy of a Stock object.
4   */
5
6  public class ObjectCopy
7  {
8      public static void main(String[] args)
9      {
10         // Create a Stock object.
11         Stock company1 = new Stock("XYZ", 9.62);
12
13         // Declare a Stock variable.
14         Stock company2;
15
16         // Make company2 reference a copy of the object
17         // referenced by company1.
18         company2 = company1.copy();
19
20         // Display the contents of both objects.
21         System.out.println("Company 1:\n" + company1);
22         System.out.println();
23         System.out.println("Company 2:\n" + company2);
24
25         // Confirm that we actually have two objects.
26         if (company1 == company2)
27         {
28             System.out.println("The company1 and company2 " +
29                               "variables reference the same obj");
29
30         }
31         else
32         {
```

```
33         System.out.println("The company1 and company2 " +
34                         "variables reference different ob
35     }
36 }
37 }
```

Program Output

Company 1:
Trading symbol: XYZ
Share price: 9.62

Company 2:
Trading symbol: XYZ
Share price: 9.62
The company1 and company2 variables reference different objects.

Copy Constructors

Another way to create a copy of an object is to use a copy constructor. A *copy constructor* is simply a constructor that accepts an object of the same class as an argument. It makes the object that is being created a copy of the object that was passed as an argument.

If you have downloaded the book's source code, in the folder [Chapter 06\Stock Class Phase 4](#), you will find another revision of the Stock class. This version of the class has a copy constructor. The code for the copy constructor follows. (No other part of the class has changed, so only the copy constructor is shown.)

```
public Stock(Stock object2)
{
    symbol = object2.symbol;
    sharePrice = object2.sharePrice;
}
```

Notice that the constructor accepts a Stock object as an argument. The parameter variable object2 will reference the object passed as an argument. The constructor copies the values in object2's symbol and sharePrice fields

to the `symbol` and `sharePrice` fields of the object being created respectively.

The following code segment demonstrates the copy constructor. It creates a `Stock` object referenced by the variable `company1`. Then it creates another `Stock` object referenced by the variable `company2`. The object referenced by `company2` is a copy of the object referenced by `company1`.

```
// Create a Stock object.  
Stock company1 = new Stock("XYZ", 9.62);  
// Create another Stock object that is a copy of the company1 obj  
Stock company2 = new Stock(company1);
```

6.9 Aggregation

Concept:

Aggregation occurs when an instance of a class is a field in another class.



VideoNote Aggregation

In real life, objects are frequently made of other objects. A house, for example, is made of door objects, window objects, wall objects, and many more. The combination of all these objects makes a house object.

When designing software, it sometimes makes sense to create an object from other objects. For example, suppose you need an object to represent a course that you are taking in college. You decide to create a `Course` class, which will hold the following information:

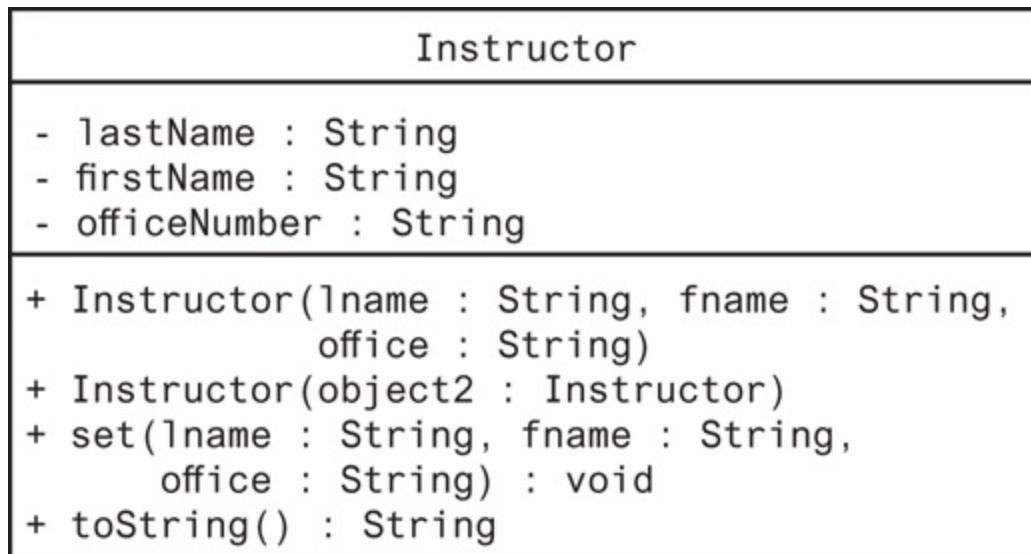
- The course name
- The instructor's last name, first name, and office number
- The textbook's title, author, and publisher

In addition to the course name, the class will hold items related to the instructor and the textbook. You could put fields for each of these items in the `Course` class. However, a good design principle is to separate related items into their own classes. In this example, an `Instructor` class could be created to hold the instructor-related data, and a `TextBook` class could be created to hold the textbook-related data. Instances of these classes could then be used as fields in the `Course` class.

Let's take a closer look at how this might be done. [Figure 6-11](#) shows a UML diagram for the `Instructor` class. To keep things simple, the class has only the following methods:

- A constructor that accepts arguments for the instructor's last name, first name, and office number
- A copy constructor
- A set method that can be used to set all of the class's fields
- A `toString` method

Figure 6-11 UML diagram for the `Instructor` class



[Figure 6-11 Full Alternative Text](#)

The code for the `Instructor` class is shown in [Code Listing 6-23](#).

Code Listing 6-23

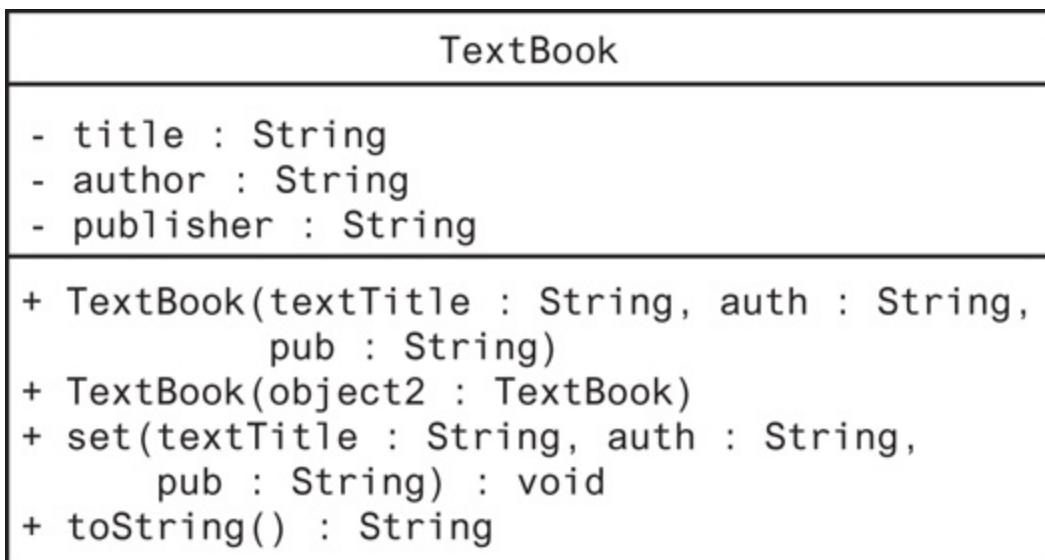
(Instructor.java)

```
1  /**
2   * This class stores information about an instructor.
3   */
4
5  public class Instructor
6  {
7      private String lastName,           // Last name
8                      firstName,        // First name
9                      officeNumber; // Office number
10
11     /**
12      * This constructor accepts arguments for the
13      * last name, first name, and office number.
14      */
15
16     public Instructor(String lname, String fname,
17                        String office)
18     {
19         lastName = lname;
20         firstName = fname;
21         officeNumber = office;
22     }
23
24     /**
25      * Copy constructor
26      */
27
28     public Instructor(Instructor object2)
29     {
30         lastName = object2.lastName;
31         firstName = object2.firstName;
32         officeNumber = object2.officeNumber;
33     }
34
35     /**
36      * The set method sets each field.
37      */
38
39     public void set(String lname, String fname,
40                     String office)
41     {
42         lastName = lname;
43         firstName = fname;
44         officeNumber = office;
```

```
45    }
46
47    /**
48     * The toString method returns a string containing
49     * the instructor information.
50     */
51
52    public String toString()
53    {
54        // Create a string representing the object.
55        String str = "Last Name: " + lastName +
56                    "\nFirst Name: " + firstName +
57                    "\nOffice Number: " + officeNumber;
58
59        // Return the string.
60        return str;
61    }
62 }
```

[Figure 6-12](#) shows a UML diagram for the `TextBook` class. As before, we want to keep the class simple. The only methods it has are a constructor, a copy constructor, a set method, and a `toString` method. The code for the `TextBook` class is shown in [Code Listing 6-24](#).

Figure 6-12 UML diagram for the TextBook class



[Figure 6-12 Full Alternative Text](#)

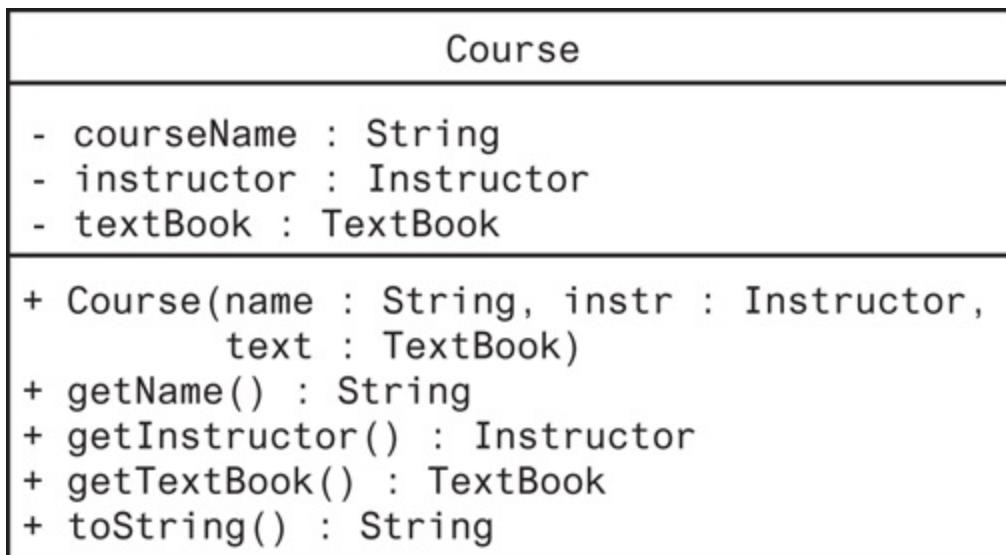
Code Listing 6-24 (TextBook.java)

```
1 /**
2  * This class stores information about a textbook.
3 */
4
5 public class TextBook
6 {
7     private String title,      // Title of the book
8                     author,    // Author's last name
9                     publisher; // Name of publisher
10
11    /**
12     * This constructor accepts arguments for the
13     * title, author, and publisher.
14     */
15
16    public TextBook(String textTitle, String auth,
```

```
17             String pub)
18     {
19         title = textTitle;
20         author = auth;
21         publisher = pub;
22     }
23
24 /**
25 * Copy constructor
26 */
27
28 public TextBook(TextBook object2)
29 {
30     title = object2.title;
31     author = object2.author;
32     publisher = object2.publisher;
33 }
34
35 /**
36 * The set method sets each field.
37 */
38
39 public void set(String textTitle, String auth,
40                  String pub)
41 {
42     title = textTitle;
43     author = auth;
44     publisher = pub;
45 }
46
47 /**
48 * The toString method returns a string containing
49 * the textbook information.
50 */
51
52 public String toString()
53 {
54     // Create a string representing the object.
55     String str = "Title: " + title +
56                 "\nAuthor: " + author +
57                 "\nPublisher: " + publisher;
58
59     // Return the string.
60     return str;
61 }
62 }
```

[Figure 6-13](#) shows a UML diagram for the Course class. Notice that the Course class has an Instructor object and a TextBook object as fields. Making an instance of one class a field in another class is called object aggregation. The word *aggregate* means “a whole that is made of constituent parts.” In this example, the Course class is an aggregate class because it is made of constituent objects.

Figure 6-13 UML diagram for the Course class



[Figure 6-13 Full Alternative Text](#)

When an instance of one class is a member of another class, it is said that there is a “has a” relationship between the classes. For example, the relationships that exist among the Course, Instructor, and TextBook classes can be described as follows:

- The course *has an* instructor.
- The course *has a* textbook.

The “has a” relationship is sometimes called a *whole–part relationship*

because one object is part of a greater whole. The code for the Course class is shown in [Code Listing 6-25](#).

Code Listing 6-25 (Course.java)

```
1  /**
2   * This class stores information about a course.
3   */
4
5  public class Course
6  {
7      private String courseName;      // Name of the course
8      private Instructor instructor; // The instructor
9      private TextBook textbook;     // The textbook
10
11     /**
12      * This constructor accepts arguments for the
13      * course name, instructor, and textbook.
14     */
15
16     public Course(String name, Instructor instr,
17                   TextBook text)
18     {
19         // Assign the courseName.
20         courseName = name;
21
22         // Create a new Instructor object, passing
23         // instr as an argument to the copy constructor.
24         instructor = new Instructor(instr);
25
26         // Create a new TextBook object, passing
27         // text as an argument to the copy constructor.
28         textbook = new TextBook(text);
29     }
30
31     /**
32      * getName method
33     */
34
35     public String getName()
36     {
37         return courseName;
38     }
39
```

```

40     /**
41      * getInstructor method
42      */
43
44     public Instructor getInstructor()
45     {
46         // Return a copy of the instructor object.
47         return new Instructor(instructor);
48     }
49
50     /**
51      * getTextBook method
52      */
53
54     public TextBook getTextBook()
55     {
56         // Return a copy of the textbook object.
57         return new TextBook(textBook);
58     }
59
60     /**
61      * The toString method returns a string containing
62      * the course information.
63      */
64
65     public String toString()
66     {
67         // Create a string representing the object.
68         String str = "Course name: " + courseName +
69                         "\nInstructor Information:\n" +
70                         instructor +
71                         "\nTextbook Information:\n" +
72                         textbook;
73
74         // Return the string.
75         return str;
76     }
77 }
```

The program in [Code Listing 6-26](#) demonstrates the Course class.

Code Listing 6-26 (CourseDemo.java)

```

1  /**
2   * This program demonstrates the Course class.
3   */
4
5 public class CourseDemo
6 {
7     public static void main(String[] args)
8     {
9         // Create an Instructor object.
10        Instructor myInstructor =
11            new Instructor("Kramer", "Shawn", "RH3010");
12
13        // Create a TextBook object.
14        TextBook myTextBook =
15            new TextBook("Starting Out with Java",
16                         "Gaddis", "Pearson");
17
18        // Create a Course object.
19        Course myCourse =
20            new Course("Intro to Java", myInstructor,
21                       myTextBook);
22
23        // Display the course information.
24        System.out.println(myCourse);
25    }
26 }

```

Program Output

```

Course name: Intro to Java
Instructor Information:
Last Name: Kramer
First Name: Shawn
Office Number: RH3010
Textbook Information:
Title: Starting Out with Java
Author: Gaddis
Publisher: Pearson

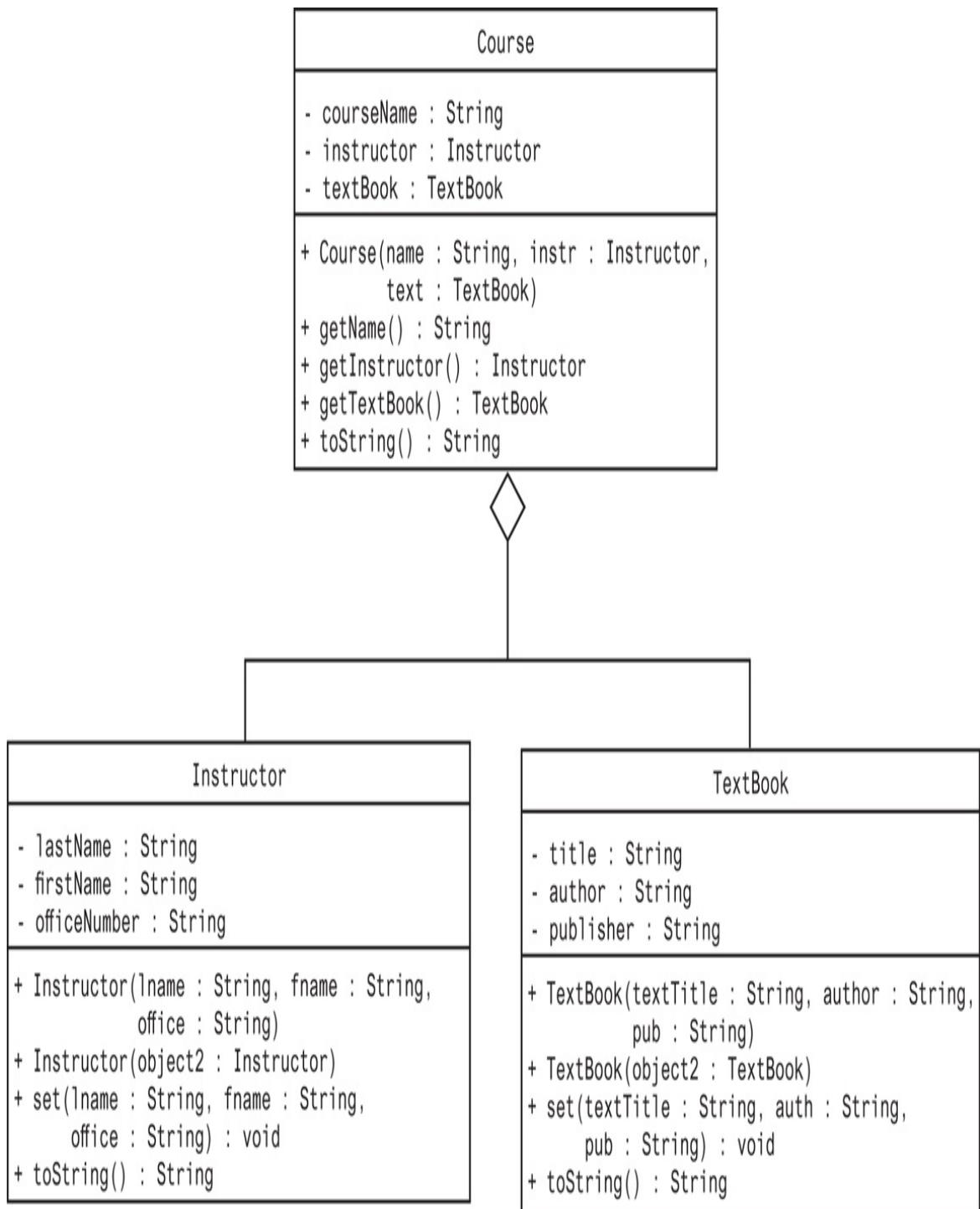
```

Aggregation in UML Diagrams

You show aggregation in a UML diagram by connecting two classes with a line that has an open diamond at one end. The diamond is closest to the class

that is the aggregate. [Figure 6-14](#) shows a UML diagram depicting the relationship among the Course, Instructor, and TextBook classes. The open diamond is closest to the course class because it is the aggregate (the whole).

Figure 6-14 UML diagram showing aggregation



[Figure 6-14 Full Alternative Text](#)

Security Issues with Aggregate

Classes

When writing an aggregate class, you should be careful not to unintentionally create “security holes” that can allow code outside the class to modify private data inside the class. We will focus on two specific practices that can help prevent security holes in your classes:

- Perform Deep Copies When Creating Field Objects

An aggregate object contains references to other objects. When you make a copy of the aggregate object, it is important that you also make copies of the objects it references. This is known as a *deep copy*. If you make a copy of an aggregate object, but only make a reference copy of the objects it references, then you have performed a *shallow copy*.

- Return Copies of Field Objects, Not the Original Objects

When a method in the aggregate class returns a reference to a field object, return a reference to a copy of the field object.

Let’s discuss each of these practices in more depth.

Perform Deep Copies When Creating Field Objects

Let’s take a closer look at the `Course` class in [Code Listing 6-25](#). First, notice the arguments that the constructor accepts in lines 16 and 17:

- A reference to a `String` containing the name of the course is passed into the `name` parameter.
- A reference to an `Instructor` object is passed into the `instr` parameter.
- A reference to a `TextBook` object is passed into the `text` parameter.

Next, notice the constructor does not merely assign `instr` to the `instructor` field. Instead, in line 24 it creates a new `Instructor` object for the `instructor` field, and passes `instr` to the copy constructor. Here is the statement:

```
instructor = new Instructor(instr);
```

This statement creates a copy of the object referenced by `instr`. The `instructor` field will reference the copy.

When a class has a field that is an object, it is possible a shallow copy operation will create a security hole. For example, suppose the `Course` constructor had been written like this:

```
// Bad constructor!
public Course(String name, Instructor instr, TextBook text)
{
    // Assign the courseName.
    courseName = name;
    // Assign the instructor (Reference copy)
    instructor = instr; // Causes security hole!
    // Assign the textbook (Reference copy)
    textbook = text; // Causes security hole!
}
```

In this example, the `instructor` and `textbook` fields are merely assigned the addresses of the objects passed into the constructor. This can cause problems because there may be variables outside the `Course` object that also contain references to these `Instructor` and `TextBook` objects. These outside variables would provide direct access to the `Course` object's private data.

At this point, you might be wondering why a deep copy was not also done for the `courseName` field. In line 20, the `Course` constructor performs a reference copy, simply assigning the address of the `String` object referenced by `name` to the `courseName` field. This is permissible because `String` objects are immutable. An immutable object does not provide a way to change its contents. Even if variables outside the `Course` class reference the same object that `courseName` references, the object cannot be changed.

Return Copies of Field Objects, Not the Original Objects

When a method in an aggregate class returns a reference to a field object, it should return a reference to a copy of the field object, not the field object itself. For example, look at the `getInstructor` method in the `Course` class. The code is shown here:

```
public Instructor getInstructor()
{
    // Return a copy of the instructor object.
    return new Instructor(instructor);
}
```

Notice the return statement uses the `new` key word to create a new `Instructor` object, passing the `instructor` field to the copy constructor. The object that is created is a copy of the object referenced by `instructor`. The address of the copy is then returned. This is preferable to simply returning a reference to the field object itself. For example, suppose the method had been written this way:

```
// Bad method
public Instructor getInstructor()
{
    // Return a reference to the instructor object.
    return instructor; // WRONG! Causes a security hole.
}
```

This method returns the value stored in the `instructor` field, which is the address of an `Instructor` object. Any variable that receives the address can then access the `Instructor` object. This means that code outside the `Course` object can change the values held by the `Instructor` object. This is a security hole because the `Instructor` object is a private field! Only code inside the `Course` class should be allowed to access it.



Note:

It is permissible to return a reference to a `String` object, even if the `String` object is a private field. This is because `String` objects are immutable.

Avoid Using null References

Recall from [Chapter 3](#) that, by default, a reference variable that is an instance field is initialized to the value `null`. This indicates that the variable does not reference an object. Because a `null` reference variable does not reference an object, you cannot use it to perform an operation that would require the existence of an object. For example, a `null` reference variable cannot be used to call a method. If you attempt to perform an operation with a `null` reference variable, the program will terminate. For example, look at the `FullName` class in [Code Listing 6-27](#).

Code Listing 6-27 (FullName.java)

```
1  /**
2   * This class stores a person's first, last, and middle names
3   * The class is dangerous because it does not prevent operations
4   * on null reference fields.
5  */
6
7 public class FullName
8 {
9     private String lastName,    // To hold a last name
10        firstName,    // To hold a first name
11        middleName; // To hold a middle name
12
13 /**
14  * The following method sets the lastName field.
15 */
16
17 public void setLastName(String str)
18 {
19     lastName = str;
20 }
21
22 /**
23  * The following method sets the firstName field.
24 */
```

```

24     */
25
26     public void setFirstName(String str)
27     {
28         firstName = str;
29     }
30
31     /**
32      * The following method sets the middleName field.
33      */
34
35     public void setMiddleName(String str)
36     {
37         middleName = str;
38     }
39
40     /**
41      * The following method returns the length of the
42      * full name.
43      */
44
45     public int getLength()
46     {
47         return lastName.length() + firstName.length() +
48                middleName.length();
49     }
50
51     /**
52      * The following method returns the full name.
53      */
54
55     public String toString()
56     {
57         return firstName + " " + middleName + " " +
58                lastName;
59     }
60 }
```

First, notice the class has three `String` reference variables as fields: `lastName`, `firstName`, and `middleName`. Second, notice the class does not have a programmer-defined constructor. When an instance of this class is created, the `lastName`, `firstName`, and `middleName` fields will be initialized to `null` by the default constructor. Third, notice the `getLength` method uses the `lastName`, `firstName`, and `middleName` variables to call the `String` class's `length` method in lines 47 and 48. Nothing is preventing the `length`

method from being called while any or all of these reference variables are set to `null`. The program in [Code Listing 6-28](#) demonstrates this.

Code Listing 6-28 (`NameTester.java`)

```
1  /**
2   * This program creates a FullName object, then calls the
3   * object's getLength method before values are established fo
4   * its reference fields. As a result, this program will crash
5   */
6
7 public class NameTester
8 {
9     public static void main(String[] args)
10    {
11        // Create a FullName object.
12        FullName name = new FullName();
13
14        // Display the length of the name.
15        System.out.println(name.getLength());
16    }
17 }
```

This program will crash¹ when you run it because the `getLength` method is called before the `name` object's fields are made to reference `String` objects. One way to prevent the program from crashing is to use `if` statements in the `getLength` method to determine whether any of the fields are set to `null`. Here is an example:

¹Actually, the program throws an exception. Exceptions will be discussed in [Chapter 10](#).

```
public int getLength()
{
    int len = 0;

    if (lastName != null)
        len += lastName.length();

    if (firstName != null)
```

```
    len += firstName.length();

    if (middleName != null)
        len += middleName.length();

    return len;
}
```

Another way to handle this problem is to write a no-arg constructor that assigns values to the reference fields. Here is an example:

```
public FullName()
{
    lastName = "";
    firstName = "";
    middleName = "";
}
```



Checkpoint

1. 6.12 Consider the following statement: “A car has an engine.” If this statement refers to classes, what is the aggregate class?
2. 6.13 Why is it not safe to return a reference to an object that is a private field? Does this also hold true for `String` objects that are private fields? Why or why not?
3. 6.14 A class has a reference variable as an instance field. Is it advisable to use the reference variable to call a method prior to assigning it the address of an object? Why or why not?

6.10 The `this` Reference Variable Concept:

The `this` key word is the name of a reference variable that an object can use to refer to itself. It is available to all nonstatic methods.

The key word `this` is the name of a reference variable that an object can use to refer to itself. For example, recall the `Stock` class presented earlier in this chapter. The class has the following `equals` method that compares the calling `Stock` object to another `Stock` object that is passed as an argument:

```
public boolean equals(Stock object2)
{
    boolean status;

    // Determine whether this object's symbol and
    // sharePrice fields are equal to object2's
    // symbol and sharePrice fields.
    if (symbol.equals(object2.symbol) &&
        sharePrice == object2.sharePrice)
        status = true; // Yes, the objects are equal.
    else
        status = false; // No, the objects are not equal.

    // Return the value in status.
    return status;
}
```

When this method is executing, the `this` variable contains the address of the calling object. We could rewrite the `if` statement as follows, and it would perform the same operation (the changes appear in bold):

```
if (this.symbol.equals(object2.symbol) &&
    this.sharePrice == object2.sharePrice)
```

The `this` reference variable is available to all of a class's nonstatic methods.

Using `this` to Overcome Shadowing

One common use of the `this` key word is to overcome the shadowing of a field name by a parameter name. Recall from [Chapter 3](#) that if a method's parameter has the same name as a field in the same class, then the parameter name shadows the field name. For example, look at the constructor in the `Stock` class:

```
public Stock(String sym, double price)
{
    symbol = sym;
    sharePrice = price;
}
```

This method uses the parameter `sym` to accept an argument assigned to the `symbol` field, and the parameter `price` to accept an argument assigned to the `sharePrice` field. Sometimes it is difficult (and even time-consuming) to think of a good parameter name that is different from a field name. To avoid this problem, many programmers give parameters the same names as the fields to which they correspond, then use the `this` key word to refer to the field names. For example the `Stock` class's constructor could be written as follows:

```
public Stock(String symbol, double sharePrice)
{
    this.symbol = symbol;
    this.sharePrice = sharePrice;
}
```

Although the parameter names `symbol` and `sharePrice` shadow the field names `symbol` and `sharePrice`, the `this` key word overcomes the shadowing. Because `this` is a reference to the calling object, the expression `this.symbol` refers to the calling object's `symbol` field, and the expression `this.sharePrice` refers to the calling object's `sharePrice` field.

Using this to Call an Overloaded Constructor from Another Constructor

You learned in [Chapter 3](#) that a constructor is automatically called when an object is created. You also learned that you cannot call a constructor explicitly, as you do other methods. However, there is one exception to this rule: You can use the `this` key word to call one constructor from another constructor in the same class.

To illustrate this, recall the `Stock` class presented earlier in this chapter. It has the following constructor:

```
public Stock(String sym, double price)
{
    symbol = sym;
    sharePrice = price;
}
```

This constructor accepts arguments that are assigned to the `symbol` and `sharePrice` fields. Let's suppose we also want a constructor that only accepts an argument for the `symbol` field, and assigns 0.0 to the `sharePrice` field. Here's one way to write the constructor:

```
public Stock(String sym)
{
    this(sym, 0.0);
}
```

This constructor simply uses the `this` variable to call the first constructor. It passes the value in `sym` as the first argument, and 0.0 as the second argument. The result is that the `symbol` field is assigned the value in `sym`, and the `sharePrice` field is assigned 0.0.

Remember the following rules about using `this` to call a constructor:

- `this` can only be used to call a constructor from another constructor in

the same class.

- It *must* be the first statement in the constructor that is making the call. If it is not the first statement, a compiler error will result.



Checkpoint

1. 6.15 Look at the following code. (You might want to review the Stock class presented earlier in this chapter.)

```
Stock stock1 = new Stock("XYZ", 9.65);
Stock stock2 = new Stock("SUNW", 7.92);
```

While the equals method is executing as a result of the following statement, what object does this reference?

```
if (stock2.equals(stock1))
    System.out.println("The stocks are the same.");
```

6.11 Inner Classes

Concept:

An inner class is a class that is defined inside another class definition.

All of the classes you have written so far have been stored separately in their own source files. Java also allows you to write a class definition inside of another class definition. A class that is defined inside of another class is called an *inner class*.² [Code Listing 6-29](#) shows an example of a class with an inner class. The program in [Code Listing 6-30](#) demonstrates the classes.

² When the class defined inside another class is written with the static modifier, it is known as a nested class, not an inner class. We do not discuss nested classes in this book.

Code Listing 6-29 (RetailItem.java)

```
1  /**
2   * This class uses an inner class.
3   */
4
5 public class RetailItem
6 {
7     private String description;    // Item description
8     private int itemNumber;       // Item number
9     private CostData cost;        // Cost data
10
11    /**
12     * RetailItem class constructor
13     */
14
```

```
15     public RetailItem(String desc, int itemNum,
16                         double wholesale, double retail)
17     {
18         description = desc;
19         itemNumber = itemNum;
20         cost = new CostData(wholesale, retail);
21     }
22
23     /**
24      * RetailItem class toString method
25      */
26
27     public String toString()
28     {
29         String str; // To hold a descriptive string.
30
31         // Create a formatted string describing the item.
32         str = String.format("Description: %s\n" +
33                             "Item Number: %d\n" +
34                             "Wholesale Cost: $%,.2f\n" +
35                             "Retail Price: $%,.2f\n",
36                             description, itemNumber,
37                             cost.wholesale, cost.retail);
38
39         // Return the string.
40         return str;
41     }
42
43     /**
44      * CostData Inner Class
45      */
46
47     private class CostData
48     {
49         public double wholesale, // Wholesale cost
50                         retail; // Retail price
51
52         /**
53          * CostData class constructor
54          */
55
56         public CostData(double w, double r)
57         {
58             wholesale = w;
59             retail = r;
60         }
61     }
62 }
```

Code Listing 6-30 (InnerClassDemo.java)

```
1  /**
2   * This program demonstrates the RetailItem class,
3   * which has an inner class.
4   */
5
6  public class InnerClassDemo
7  {
8      public static void main(String[] args)
9      {
10         // Create a RetailItem object.
11         RetailItem item = new RetailItem("Candy bar", 17789,
12                                         0.75, 1.5);
13
14         // Display the item's information.
15         System.out.println(item);
16     }
17 }
```

Program Output

```
Description: Candy bar
Item Number: 17789
Wholesale Cost: $0.75
Retail Price: $1.50
```

The `RetailItem` class is an aggregate class. It has as a field an instance of the `CostData` class. Notice the `CostData` class is defined inside of the `RetailItem` class. The `RetailItem` class is the outer class, and the `CostData` class is the inner class.

An inner class is visible only to code inside the outer class. This means the use of the inner class is restricted to the outer class. Only code in the outer class may create an instance of the inner class.

One unusual aspect of the `CostData` class is that its fields, `wholesale` and

`retail`, are declared as `public`. Although [Chapter 3](#) warns against making a field `public`, it is permissible in the case of inner classes. This is because the inner class's members are not accessible to code outside the outer class. Even though the `CostData` class's fields are `public`, only code in the

`RetailItem` class can access its members. In effect, the `CostData` class's `public` members are like

private members of the `RetailItem` class. The following points summarize the accessibility issues between inner and outer classes:

- An outer class can access the public members of an inner class.
- A private inner class is not visible or accessible to code outside the outer class.
- An inner class can access the private members of the outer class.

Although you will not write inner classes very often, you can use them to create classes that are visible and accessible only to specific other classes. Later in this book, we will use inner classes in graphics programs.



Note:

When a class with an inner class is compiled, byte code for the inner class will be stored in a separate file. The file's name will consist of the name of the outer class, followed by a `$` character, followed by the name of the inner class, followed by `.class`. For example, the byte code for the `CostData` class in [Code Listing 6-29](#) would be stored in the file

`RetailItem$CostData.class`.

6.12 Enumerated Types

Concept:

An enumerated data type consists of a set of predefined values. You can use the data type to create variables that can hold only the values that belong to the enumerated data type.

You've already learned the concept of data types and how they are used with primitive variables. For example, a variable of the `int` data type can hold integer values within a certain range. You cannot assign floating-point values to an `int` variable because only `int` values may be assigned to `int` variables. A data type defines the values that are legal for any variable of that data type.

Sometimes, it is helpful to create your own data type that has a specific set of legal values. For example, suppose you wanted to create a data type named `Day`, and the legal values in that data type were the names of the days of the week (Sunday, Monday, and so on). When you create a variable of the `Day` data type, you can only store the names of the days of the week in that variable. Any other values would be illegal. In Java, such a type is known as an *enumerated data type*.

You use the `enum` key word to create your own data type and specify the values that belong to that type. Here is an example of an enumerated data type declaration:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
          THURSDAY, FRIDAY, SATURDAY }
```

An enumerated data type declaration begins with the key word `enum`, followed by the name of the type, followed by a list of identifiers inside braces. The example declaration creates an enumerated data type named `Day`. The identifiers `SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, and

SATURDAY, which are listed inside the braces, are known as enum *constants*. They represent the values that belong to the Day data type. Here is the general format of an enumerated type declaration:

```
enum TypeName { One or more enum constants }
```

Note that the enum constants are not enclosed in quotation marks; therefore, they are not strings. enum constants must be legal Java identifiers.

Tip:

When making up names for enum constants, it is not required that they be written in all uppercase letters. We could have written the Day type's enum constants as sunday, monday, and so forth. Because they represent constant values, however, the standard convention is to write them in all uppercase letters.

Once you have created an enumerated data type in your program, you can declare variables

of that type. For example, the following statement declares workDay as a variable of the Day type:

```
Day workDay;
```

Because workDay is a Day variable, the only values that we can legally assign to it are the enum constants Day.SUNDAY, Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY, and Day.SATURDAY. If we try to assign any value other than one of the Day type's enum constants, a compiler error will result. For example, the following statement assigns the value Day.WEDNESDAY to the workDay variable.

```
Day workDay = Day.WEDNESDAY;
```

Notice that we assigned Day.WEDNESDAY instead of just WEDNESDAY. The name Day.WEDNESDAY is the *fully qualified name* of the Day type's WEDNESDAY constant. Under most circumstances, you must use the fully qualified name of

an enum constant.

Enumerated Types Are Specialized Classes

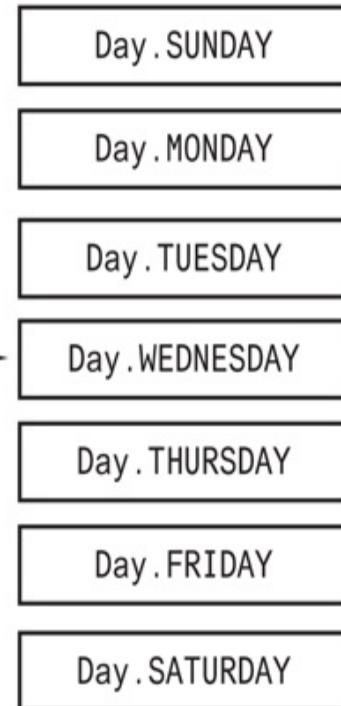
When you write an enumerated type declaration, you are actually creating a special kind of class. In addition, the enum constants you list inside the braces are actually objects of the class. In the previous example, Day is a class, and the enum constants Day .SUNDAY, Day .MONDAY, Day .TUESDAY, Day .WEDNESDAY, Day .THURSDAY, Day .FRIDAY, and Day .SATURDAY are all instances of the Day class. When we assigned Day .WEDNESDAY to the workDay variable, we were assigning the address of the Day .WEDNESDAY object to the variable. This is illustrated in [Figure 6-15](#).

Figure 6-15 The workDay variable references the Day .WEDNESDAY object

Each of these are objects of the Day type, which is a specialized class.

The workDay variable holds the address of the Day .WEDNESDAY object.

address



[Figure 6-15 Full Alternative Text](#)

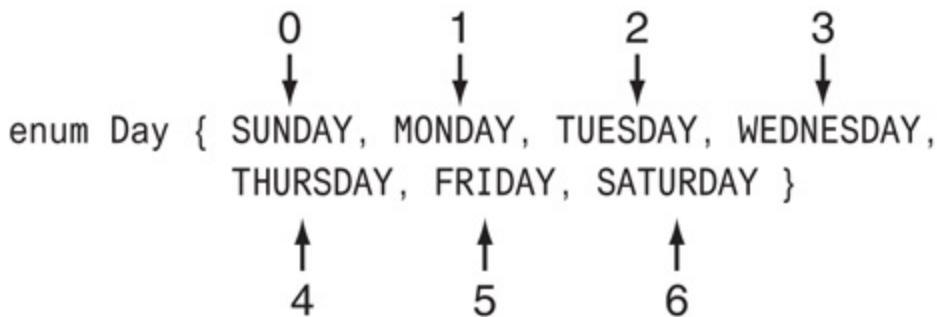
enum constants, which are actually objects, come automatically equipped with a few methods. One of them is the `toString` method. The `toString` method simply returns the name of the calling enum constant as a string. For example, assuming that the `Day` type has been declared as previously shown, both of the following code segments display the string `WEDNESDAY`. (Recall that the `toString` method is implicitly called when an object is passed to `System.out.println`.)

```
// This code displays WEDNESDAY.  
Day workDay = Day.WEDNESDAY;  
System.out.println(workDay);  
  
// This code also displays WEDNESDAY.  
System.out.println(Day.WEDNESDAY);
```

enum constants also have a method named `ordinal`. The `ordinal` method returns an integer value representing the constant's ordinal value. The

constant's *ordinal value* is its position in the enum declaration, with the first constant being at position 0. [Figure 6-16](#) shows the ordinal values of each of the constants declared in the Day data type.

Figure 6-16 The Day enumerated data type and the ordinal positions of its enum constants



For example, assuming that the Day type has been declared as previously shown, look at the following code segment:

```
Day lastWorkDay = Day.FRIDAY;
System.out.println(lastWorkDay.ordinal());
System.out.println(Day.MONDAY.ordinal());
```

The ordinal value for Day.FRIDAY is 5, and the ordinal value for Day.MONDAY is 1, so this code will display:

5
1

The last enumerated data type methods we will discuss here are equals and compareTo. The equals method accepts an object as its argument and returns true if that object is equal to the calling enum constant. For example, assuming that the Day type has been declared as previously shown, the

following code segment will display “The two are the same”:

```
Day myDay = Day.TUESDAY;
if (myDay.equals(Day.TUESDAY))
    System.out.println("The two are the same.");
```

The `compareTo` method is designed to compare `enum` constants of the same type. It accepts an object as its argument and returns

- a negative integer value if the calling `enum` constant’s ordinal value is less than the argument’s ordinal value.
- zero if the calling `enum` constant is the same as the argument.
- a positive integer value if the calling `enum` constant’s ordinal value is greater than the argument’s ordinal value.

For example, assuming the `Day` type has been declared as previously shown, the following code segment will display “FRIDAY is greater than MONDAY”:

```
Day myDay = Day.FRIDAY;
if (myDay.compareTo(Day.MONDAY) > 0)
    System.out.println(myDay + " is greater than " +
        Day.MONDAY);
```

One place to declare an enumerated type is inside a class. If you declare an enumerated type inside a class, it cannot be inside a method. [Code Listing 6-31](#) shows an example. It demonstrates the `Day` enumerated type.

Code Listing 6-31 (EnumDemo.java)

```
1 /**
2  * This program demonstrates an enumerated type.
3 */
4
5 public class EnumDemo
6 {
7     // Declare the Day enumerated type.
8     enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
```

```

9             THURSDAY, FRIDAY, SATURDAY }
10
11    public static void main(String[] args)
12    {
13        // Declare a Day variable and assign it a value.
14        Day workDay = Day.WEDNESDAY;
15
16        // The following statement displays WEDNESDAY.
17        System.out.println(workDay);
18
19        // The following statement displays the ordinal
20        // value for Day.SUNDAY, which is 0.
21        System.out.println("The ordinal value for " +
22                            Day.SUNDAY + " is " +
23                            Day.SUNDAY.ordinal());
24
25        // The following statement displays the ordinal
26        // value for Day.SATURDAY, which is 6.
27        System.out.println("The ordinal value for " +
28                            Day.SATURDAY + " is " +
29                            Day.SATURDAY.ordinal());
30
31        // The following statement compares two enum constants.
32        if (Day.FRIDAY.compareTo(Day.MONDAY) > 0)
33            System.out.println(Day.FRIDAY + " is greater than " +
34                                Day.MONDAY);
35        else
36            System.out.println(Day.FRIDAY + " is NOT greater than "
37                                Day.MONDAY);
38    }
39 }
```

Program Output

```

WEDNESDAY
The ordinal value for SUNDAY is 0
The ordinal value for SATURDAY is 6
FRIDAY is greater than MONDAY
```

You can also write an enumerated type declaration inside its own file. If you do, the file name must match the name of the type. For example, if we stored the Day type in its own file, we would name the file Day.java. This makes sense because enumerated data types are specialized classes. For example, look at [Code Listing 6-32](#). This file, CarType.java, contains the declaration

of an enumerated data type named `CarType`. When it is compiled, a byte code file named `CarType.class` will be generated.

Code Listing 6-32 (`CarType.java`)

```
1  /**
2   * CarType enumerated data type
3   */
4
5 enum CarType { PORSCHE, FERRARI, JAGUAR }
```

Also look at [Code Listing 6-33](#). This file, `CarColor.java`, contains the declaration of an enumerated data type named `CarColor`. When it is compiled, a byte code file named `CarColor.class` will be generated.

Code Listing 6-33 (`CarColor.java`)

```
1  /**
2   * CarColor enumerated data type
3   */
4
5 enum CarColor { RED, BLACK, BLUE, SILVER }
```

[Code Listing 6-34](#) shows the `SportsCar` class, which uses these enumerated types. [Code Listing 6-35](#) demonstrates the class.

Code Listing 6-34 (`SportsCar.java`)

```
1  /**
2   * SportsCar class
3   */
4
5 public class SportsCar
6 {
7     private CarType make;      // The car's make
8     private CarColor color;   // The car's color
9     private double price;    // The car's price
```

```
10
11     /**
12      * The constructor accepts arguments for the
13      * car's make, color, and price.
14      */
15
16     public SportsCar(CarType aMake, CarColor aColor,
17                        double aPrice)
18     {
19         make = aMake;
20         color = aColor;
21         price = aPrice;
22     }
23
24     /**
25      * getMake method
26      */
27
28     public CarType getMake()
29     {
30         return make;
31     }
32
33     /**
34      * getColor method
35      */
36
37     public CarColor getColor()
38     {
39         return color;
40     }
41
42     /**
43      * getPrice method
44      */
45
46     public double getPrice()
47     {
48         return price;
49     }
50
51     /**
52      * toString method
53      */
54
55     public String toString()
56     {
57         // Create a string representing the object.
```

```

58     String str = String.format("Make: %s\n" +
59             "Color: %s\n" +
60             "Price: $%,.2f",
61             make, color, price);
62     // Return the string.
63     return str;
64 }
65 }
```

Code Listing 6-35 (SportsCarDemo.java)

```

1  /**
2   * This program demonstrates the SportsCar class.
3   */
4
5  public class SportsCarDemo
6  {
7      public static void main(String[] args)
8      {
9          // Create a SportsCar object.
10         SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
11                                         CarColor.RED, 100000);
12
13         // Display the object's values.
14         System.out.println(yourNewCar);
15     }
16 }
```

Program Output

Make: PORSCHE
 Color: RED
 Price: \$100,000.00

Switching on an Enumerated Type

Java allows you to test an enum constant with a switch statement. For example, look at the program in [Code Listing 6-36](#). It creates a SportsCar object, then uses a switch statement to test the object's make field.

Code Listing 6-36 (SportsCarDemo2.java)

```
1  /**
2   * This program shows that you can switch on an
3   * enumerated type.
4   */
5
6 public class SportsCarDemo2
7 {
8     public static void main(String[] args)
9     {
10         // Create a SportsCar object.
11         SportsCar yourNewCar = new SportsCar(CarType.PORSCHE,
12                                         CarColor.RED, 100000);
13
14         // Get the car make and switch on it.
15         switch (yourNewCar.getMake())
16         {
17             case PORSCHE :
18                 System.out.println("Your car was made in Germany.");
19                 break;
20             case FERRARI :
21                 System.out.println("Your car was made in Italy.");
22                 break;
23             case JAGUAR :
24                 System.out.println("Your car was made in England.");
25                 break;
26             default:
27                 System.out.println("I'm not sure where that car "
28                                     "was made.");
29         }
30     }
31 }
```

Program Output

Your car was made in Germany.

In line 15, the switch statement tests the value returned from the `yourNewCar.getMake()` method. This method returns a `CarType` enumerated constant. Based upon the value returned from the method, the program then branches to the appropriate case statement. Notice in the case statements that

the enumerated constants are not fully qualified. In other words, we had to write PORSCHE, FERRARI, and JAGUAR instead of CarType.PORSCHE, CarType.FERRARI, and CarType.JAGUAR. If you give a fully qualified enum constant name as a case expression, a compiler error will result.

Tip:

Notice that the switch statement in [Code Listing 6-37](#) has a default section, even though it has a case statement for every enum constant in the CarType type. This will handle things in the event that more enum constants are added to the CarType file. This type of planning is an example of “defensive” programming.

Checkpoint

1. 6.16 Look at the following statement, which declares an enumerated data type.

```
enum Flower { ROSE, DAISY, PETUNIA }
```

1. What is the name of the data type?
 2. What is the ordinal value for the enum constant ROSE? For DAISY? For PETUNIA?
 3. What is the fully qualified name of the enum constant ROSE? Of DAISY? Of PETUNIA?
 4. Write a statement that declares a variable of this enumerated data type. The variable should be named flora. Initialize the variable with the PETUNIA constant.
2. 6.17 Assume the following enumerated data type has been declared.

```
enum Creatures{ HOBBIT, ELF, DRAGON }
```

What will the following code display?

```
System.out.println(Creatures.HOBBIT + " " +
                    Creatures.ELF + " " +
                    Creatures.DRAGON);
```

3. 6.18 Assume the following enumerated data type has been declared.

```
enum Letters { Z, Y, X }
```

What will the following code display?

```
if (Letters.Z.compareTo(Letters.X) > 0)
    System.out.println("Z is greater than X.");
else
    System.out.println("Z is not greater than X.");
```

6.13 Garbage Collection

Concept:

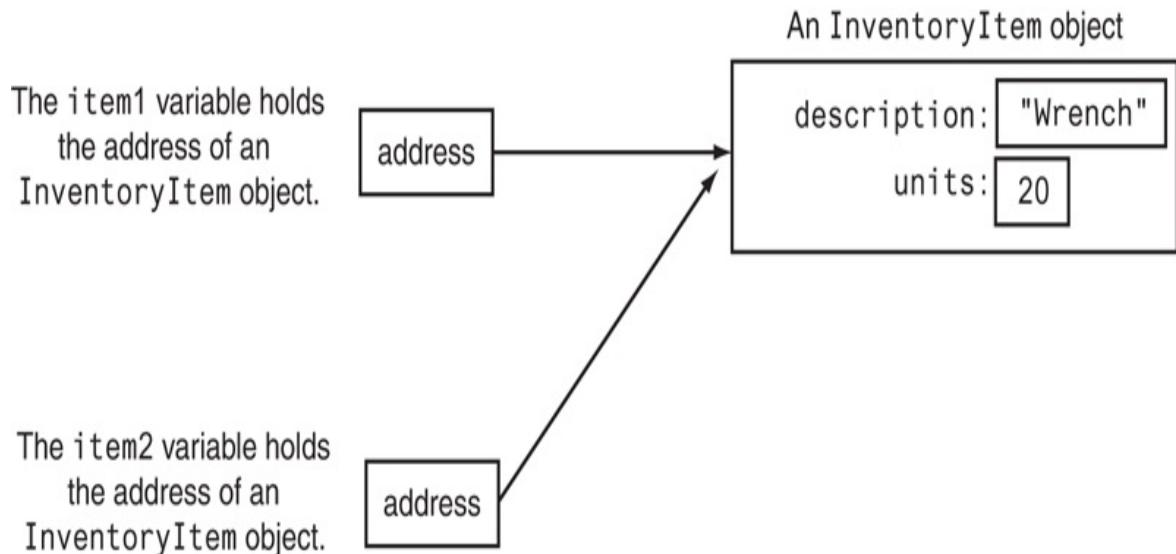
The Java Virtual Machine periodically runs a process known as the garbage collector, which removes unreferenced objects from memory.

When an object is no longer needed, it should be deleted so that the memory it uses can be freed for other purposes. Fortunately, you do not have to delete objects after you are finished using them. The JVM periodically performs a process known as garbage collection, which automatically removes unreferenced objects from memory. For example, look at the following code:

```
// Declare two InventoryItem reference variables.  
InventoryItem item1, item2;  
  
// Create an object and reference it with item1.  
item1 = new InventoryItem("Wrench", 20);  
  
// Reference the same object with item2.  
item2 = item1;  
  
// Store null in item1 so it no longer references the object.  
item1 = null;  
  
// The object is still referenced by item2, though.  
// Store null in item2 so it no longer references the object.  
item2 = null;  
  
// Now the object is no longer referenced, so it can be removed  
// by the garbage collector.
```

This code uses two reference variables, `item1` and `item2`. An `InventoryItem` object is created and referenced by `item1`. Then, `item1` is assigned to `item2`, which causes `item2` to reference the same object as `item1`. This is illustrated in [Figure 6-17](#).

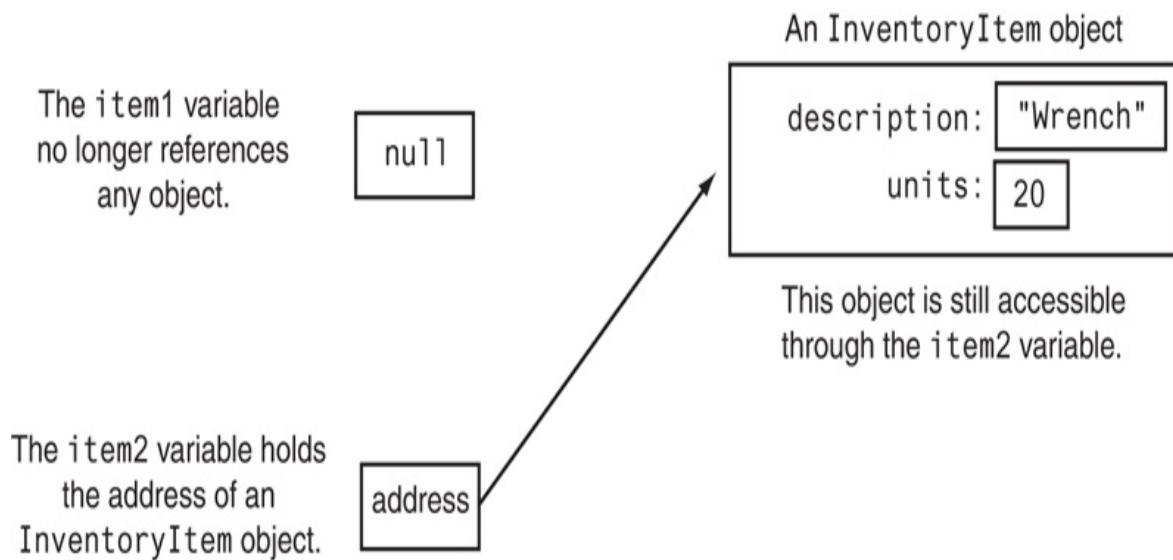
Figure 6-17 Both item1 and item2 reference the same object



[Figure 6-17 Full Alternative Text](#)

Next, the `null` value is assigned to `item1`. This removes the address of the object from the `item1` variable, causing it to no longer reference the object. [Figure 6-18](#) illustrates this.

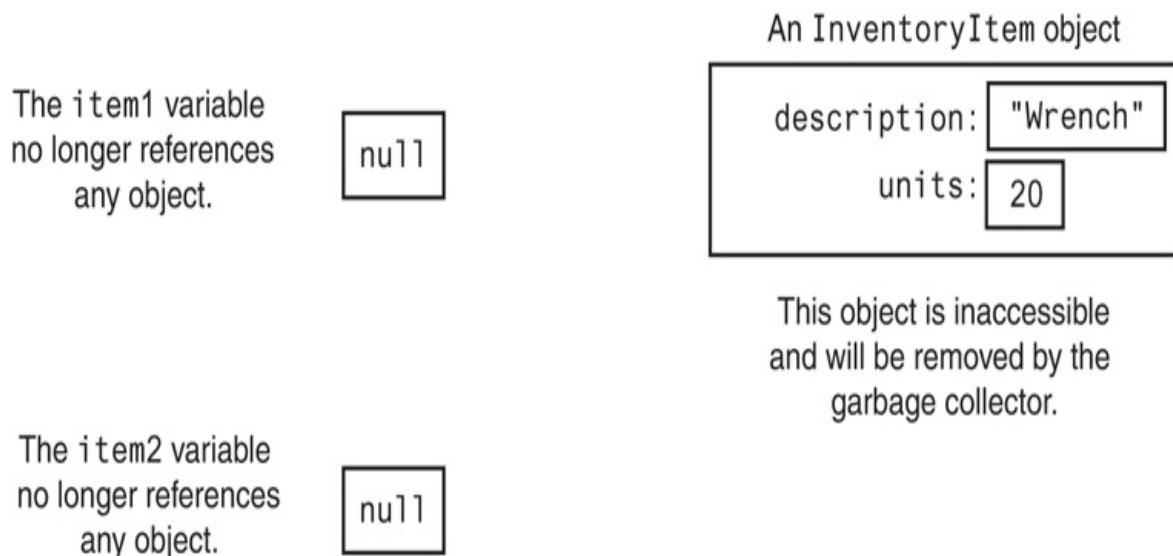
Figure 6-18 The object is only referenced by the item2 variable



[Figure 6-18 Full Alternative Text](#)

The object is still accessible, however, because it is referenced by the `item2` variable. The next statement assigns `null` to `item2`. This removes the object's address from `item2`, causing it to no longer reference the object. [Figure 6-19](#) illustrates this. Because the object is no longer accessible, it will be removed from memory the next time the garbage collector process runs.

Figure 6-19 The object is no longer referenced



[Figure 6-19 Full Alternative Text](#)

The finalize Method

If a class has a method named `finalize`, it is automatically called just before an instance of the class is deleted by the garbage collector. If you wish to execute code just before an object is deleted, you can create a `finalize` method in the class and place the code there. The `finalize` method accepts no arguments and has a `void` return type.



Note:

The garbage collector runs periodically, and you cannot predict exactly when it will execute. Therefore, you cannot know exactly when an object's `finalize` method will execute.

6.14 Focus on Object-Oriented Design: Class Collaboration

Concept:

It is common for classes to interact, or collaborate, with each other to perform their operations. Part of the object-oriented design process is identifying the collaborations between classes.

In an object-oriented application, it is common for objects of different classes to collaborate. This simply means that objects interact with each other. Sometimes, one object will need the services of another object to fulfill its responsibilities. For example, let's say an object needs to read values from the keyboard, then write those values to a file. The object would use the services of a `Scanner` object to read the values from the keyboard, then use the services of a `PrintWriter` object to write the values to a file. In this example, the object is collaborating with objects created from classes in the Java API. The objects you create from your own classes can also collaborate with each other.

If one object is to collaborate with another object, then it must know something about the other object's class methods and how to call them. For example, suppose we were to write a class named `StockPurchase`, which uses an object of the `Stock` class (presented earlier in this chapter) to simulate the purchase of a stock. The `StockPurchase` class is responsible for calculating the cost of the stock purchase. To do that, it must know how to call the `Stock` class's `getSharePrice` method to get the price per share of the stock. [Code Listing 6-37](#) shows an example of the `StockPurchase` class. (If you have downloaded the book's source code, you will find this file in the folder [*Chapter 06\StockPurchase Class.*](#))

Code Listing 6-37 (StockPurchase.java)

```
1  /**
2   * The StockPurchase class represents a stock purchase.
3   */
4
5  public class StockPurchase
6  {
7      private Stock stock; // The stock that was purchased
8      private int shares; // Number of shares owned
9
10     /**
11      * The constructor accepts arguments for the
12      * stock and number of shares.
13      */
14
15     public StockPurchase(Stock stockObject, int numShares)
16     {
17         // Create a copy of the object referenced by
18         // stockObject.
19         stock = new Stock(stockObject);
20         shares = numShares;
21     }
22
23     /**
24      * getStock method
25      */
26
27     public Stock getStock()
28     {
29         // Return a copy of the object referenced by stock.
30         return new Stock(stock);
31     }
32
33     /**
34      * getShares method
35      */
36
37     public int getShares()
38     {
39         return shares;
40     }
41
```

```

42     /**
43      * The getCost method returns the cost of the
44      * stock purchase.
45     */
46
47     public double getCost()
48     {
49         return shares * stock.getSharePrice();
50     }
51 }
```

The constructor for this class accepts a Stock object representing the stock being purchased, and an int representing the number of shares to purchase. In line 19, we see the first collaboration: the StockPurchase constructor makes a copy of the Stock object by using the Stock class's copy constructor. The copy constructor is used again in the getStock method, in line 30, to return a copy of the Stock object.

The next collaboration takes place in the getCost method. This method calculates and returns the cost of the stock purchase. In line 49, it calls the Stock class's getSharePrice method to determine the stock's price per share. The program in [Code Listing 6-38](#) demonstrates this class. (If you have downloaded the book's source code, you will find this file in the folder [*Chapter 06\StockPurchase Class*](#).)

Code Listing 6-38 (StockTrader.java)

```

1 import java.util.Scanner;
2
3 /**
4  * This program allows you to purchase shares of XYZ
5  * company's stock.
6 */
7
8 public class StockTrader
9 {
10    public static void main(String[] args)
11    {
12        int sharesToBuy; // Number of shares to buy.
```

```

13
14     // Create a Stock object for the company stock.
15     // The trading symbol is XYZ and the stock is
16     // currently $9.62 per share.
17     Stock xyzCompany = new Stock("XYZ", 9.62);
18
19     // Create a Scanner object for keyboard input.
20     Scanner keyboard = new Scanner(System.in);
21
22     // Display the current share price.
23     System.out.printf("XYZ Company's stock is currently $%,
24                         "per share.\n", xyzCompany.getSharePr
25
26     // Get the number of shares to purchase.
27     System.out.print("How many shares do you want to buy? "
28                     sharesToBuy = keyboard.nextInt();
29
30     // Create a StockPurchase object for the transaction.
31     StockPurchase buy =
32         new StockPurchase(xyzCompany, sharesToBuy);
33
34     // Display the cost of the transaction.
35     System.out.printf("Cost of the stock: $%,.2f\n", buy.ge
36 }
37 }
```

Program Output with Example Input Shown in Bold

XYZ Company's stock is currently \$9.62 per share.

How many shares do you want to buy? **100** 
Cost of the stock: \$962.00

Determining Class Collaborations with CRC Cards

During the object-oriented design process, you can determine many of the collaborations that will be necessary between classes by examining the responsibilities of the classes. In [Chapter 3](#), Section 3.7, we discussed the process of finding the classes and their responsibilities. Recall from that section that a class's responsibilities are:

- the things that the class is responsible for knowing.
- the actions that the class is responsible for doing.

Often, you will determine that the class must collaborate with another class to fulfill one or more of its responsibilities. One popular method of discovering a class's responsibilities and collaborations is by creating CRC cards. CRC stands for class, responsibilities, and collaborations.

You can use simple index cards for this procedure. Once you have gone through the process of finding the classes (which was discussed in [Chapter 3](#), Section 3.8), set aside one index card for each class. At the top of the index card, write the name of the class. Divide the rest of the card into two columns. In the left column, write each of the class's responsibilities. As you write each responsibility, think about whether the class needs to collaborate with another class to fulfill that responsibility. Ask yourself questions such as:

- Will an object of this class need to get data from another object to fulfill this responsibility?
- Will an object of this class need to request another object to perform an operation to fulfill this responsibility?

If collaboration is required, write the name of the collaborating class in the right column, next to the responsibility that requires it. If no collaboration is required for a responsibility, simply write "None" in the right column, or leave it blank. [Figure 6-20](#) shows an example CRC card for the StockPurchase class.

Figure 6-20 CRC card

Name of the class

StockPurchase		
Know the stock to purchase	Stock class	
Know the number of shares to purchase	None	
Calculate the cost of the purchase	Stock class	

Responsibilities { }

Collaborating classes }

[Figure 6-20 Full Alternative Text](#)

From the CRC card shown in the figure, we can see that the `StockPurchase` class has the following responsibilities and collaborations:

- Responsibility: To know the stock to purchase
Collaboration: The `Stock` class
- Responsibility: To know the number of shares to purchase
Collaboration: None
- Responsibility: To calculate the cost of the purchase
Collaboration: The `Stock` class

When you have completed a CRC card for each class in the application, you will have a good idea of each class's responsibilities and how the classes must interact.

6.15 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- Trying to overload methods by giving them different return types. Overloaded methods must have unique parameter lists.
- Forgetting to write a no-arg constructor for a class of which you want to be able to create instances, without passing arguments to the constructor. If you write a constructor that accepts arguments, you must also write a no-arg constructor for the same class if you want to be able to create instances of the class without passing arguments to the constructor.
- In a method that accepts an object as an argument, writing code that accidentally modifies the object. When a reference variable is passed as an argument to a method, the method has access to the object that the variable references. When writing a method that receives a reference variable as an argument, you must take care not to accidentally modify the contents of the object referenced by the variable.
- Allowing a null reference to be used. Because a `null` reference variable does not reference an object, you cannot use it to perform an operation that would require the existence of an object. For example, a `null` reference variable cannot be used to call a method. If you attempt to perform an operation with a `null` reference variable, the program will terminate. This can happen when a class has a reference variable as a field, and the variable is not properly initialized with the address of an object.
- Forgetting to use the fully qualified name of an enum constant. Under most circumstances, you must use the fully qualified name of an enum constant. One exception to this is when the `enum` constant is used as a case expression in a `switch` statement.

- Attempting to refer to an instance field or instance method in a static method. **Static** methods can refer only to other class members that are static.

Review Questions and Exercises

Multiple Choice and True/False

1. This type of method cannot access any nonstatic member variables in its own class.
 1. `instance`
 2. `void`
 3. `static`
 4. `nonstatic`
2. Two or more methods in a class may have the same name, as long as this is different.
 1. their return values
 2. their access specifier
 3. their signatures
 4. their memory address
3. The process of matching a method call with the correct method is known as .
 1. matching
 2. binding
 3. linking

4. connecting
4. When an object is passed as an argument to a method, this is actually passed.
 1. a copy of the object
 2. the name of the object
 3. a reference to the object
 4. None of these. You cannot pass an object.
5. If you write this method for a class, Java will automatically call it any time you concatenate an object of the class with a string.
 1. `toString`
 2. `plusString`
 3. `stringConvert`
 4. `concatString`
6. Making an instance of one class a field in another class is called
 - .
1. nesting
2. class fielding
3. aggregation
4. concatenation
7. This is the name of a reference variable that is always available to an instance method, and refers to the object that is calling the method.
 1. `callingObject`

2. this
 3. me
 4. instance
8. This enum method returns the position of an enum constant in the declaration.
1. position
 2. location
 3. ordinal
 4. `toString`
9. Assuming the following declaration exists:
- ```
enum Seasons { SPRING, WINTER, SUMMER, FALL }
```
- what is the fully qualified name of the FALL constant?
1. FALL
  2. enum.FALL
  3. FALL.Seasons
  4. Seasons.FALL
10. You cannot use the fully qualified name of an enum constant for this.
1. a switch expression
  2. a case expression
  3. an argument to a method
  4. all of these

11. A class that is defined inside of another class is called a(n)
- 1. inner class
  - 2. folded class
  - 3. hidden class
  - 4. unknown class
12. The JVM periodically performs this process, which automatically removes unreferenced objects from memory.
- 1. memory cleansing
  - 2. memory deallocation
  - 3. garbage collection
  - 4. object expungement
13. If a class has this method, it is called automatically just before an instance of the class is deleted by the JVM.
- 1. `finalize`
  - 2. `destroy`
  - 3. `remove`
  - 4. `housekeeper`
14. CRC stands for which of the following?
- 1. Class, Return value, Composition
  - 2. Class, Responsibilities, Collaborations

3. Class, Responsibilities, Composition
  4. Compare, Return, Continue
15. **True or False:** A static member method may refer to nonstatic member variables of the same class at any time.
16. **True or False:** All static member variables are initialized to –1 by default.
17. **True or False:** A class may not have more than one constructor.
18. **True or False:** When an object is passed as an argument to a method, the method can access the argument.
19. **True or False:** A method cannot return a reference to an object.
20. **True or False:** A private class that is defined inside another class is not visible to code outside the outer class.
21. **True or False:** You can declare an enumerated data type inside a method.
22. **True or False:** Enumerated data types are actually special types of classes.
23. **True or False:** enum constants have a `toString` method.

## Find the Error

Each of the following class definitions has errors. Find as many as you can.

1. `public class MyClass`

```
{
 private int x;
 private double y;
```

```
 public static void setValues(int a, double b)
 {
 x = a;
 y = b;
 }
}
```

2. public class TwoValues

```
{
 private int x, y;

 public TwoValues()
 {
 x = 0;
 }

 public TwoValues()
 {
 x = 0;
 y = 0;
 }
}
```

3. public class MyMath

```
{
 public static int square(int number)
 {
 return number * number;
 }

 public static double square(int number)
 {
 return number * number;
 }
}
```

4. Assume the following declaration exists:

```
enum Coffee { MEDIUM, DARK, DECAF }
```

Find the error(s) in the following switch statement:

```

// This code has errors!
Coffee myCup = DARK;
switch (myCup)
{
 case Coffee.MEDIUM :
 System.out.println("Mild flavor.");
 break;
 case Coffee.DARK :
 System.out.println("strong flavor.");
 break;
 case Coffee.DECAF :
 System.out.println("Won't keep you awake.");
 break;
 default:
 System.out.println("Never heard of it.");
}

```

# Algorithm Workbench

1. Consider the following class declaration:

```

public class Circle
{
 private double radius;

 private void getArea()
 {
 return Math.PI * radius * radius;
 }

 private double getRadius()
 {
 return radius;
 }
}

```

1. Write a no-arg constructor for this class. It should assign the `radius` field the value 0.
2. Write an overloaded constructor for this class. It should accept an argument copied into the `radius` member variable.
3. Write a `toString` method for this class. The method should return a

string containing the radius and area of the circle.

4. Write an `equals` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object contains the same data as the calling object, or `false` otherwise.
5. Write a `greaterThan` method for this class. The method should accept a `Circle` object as an argument. It should return `true` if the argument object has an area greater than the area of the calling object, or `false` otherwise.
2. A pet store sells dogs, cats, birds, and hamsters. Write a declaration for an enumerated data type that can represent the types of pets the store sells.

## Short Answer

1. Describe one thing you cannot do with a `static` method.
2. Why are `static` methods useful in creating utility classes?
3. Consider the following class declaration:

```
public class Thing
{
 private int x;
 private int y;
 private static int z = 0;

 public Thing()
 {
 x = z;
 y = z;
 }

 public static void putThing(int a)
 {
 z = a;
 }
}
```

}

Assume a program containing the class declaration defines three `Thing` objects with the following statements:

```
Thing one = new Thing();
Thing two = new Thing();
Thing three = new Thing();
```

1. How many separate instances of the `x` member exist?
2. How many separate instances of the `y` member exist?
3. How many separate instances of the `z` member exist?
4. What value will be stored in the `x` and `y` members of each object?
5. Write a statement that will call the `putThing` method.
6. When the same name is used for two or more methods in the same class, how does Java tell them apart?
7. How does method overloading improve the usefulness of a class?
8. Describe the difference in the way variables and class objects are passed as arguments to a method.
9. If you do not write an `equals` method for a class, Java provides one. Describe the behavior of the `equals` method that Java automatically provides.
10. A “has a” relationship can exist between classes. What does this mean?
11. What happens if you attempt to call a method using a reference variable that is set to `null`?
12. Is it advisable or not advisable to write a method that returns a reference to an object that is a private field? What is the exception to this?
13. What is the `this` key word?

12. Look at the following declaration:

```
enum Color { RED, ORANGE, GREEN, BLUE }
```

1. What is the name of the data type declared by this statement?
2. What are the enum constants for this type?
3. Write a statement that defines a variable of this type, and initializes it with a valid value.

13. Assuming the following enum declaration exists:

```
enum Dog { POODLE, BOXER, TERRIER }
```

What will the following statements display?

1. System.out.println(Dog.POODLE + "\n" + Dog.BOXER + "\n" + Dog.TERRIER);
2. System.out.println(Dog.POODLE.ordinal() + "\n" + Dog.BOXER.ordinal() + "\n" + Dog.TERRIER.ordinal());
3. Dog myDog = Dog.BOXER;  
    if (myDog.compareTo(Dog.TERRIER) > 0)  
        System.out.println(myDog + " is greater than " + Dog.TERRIER);  
    else  
        System.out.println(myDog + " is NOT greater than " + Dog.TERRIER);

14. Under what circumstances does an object become a candidate for garbage collection?

# Programming Challenges

## 1. Area Class

Write a class that has three overloaded static methods for calculating the areas of the following geometric shapes.

- circles
- rectangles
- cylinders

Here are the formulas for calculating the area of the shapes:

Area of a circle:  $\text{Area} = \pi r^2$

where  $\pi$  is `Math.PI` and  $r$  is the circle's radius

Area of a rectangle:  $\text{Area} = \text{width} \times \text{length}$

Area of a cylinder:  $\text{Area} = \pi r^2 h$

where  $\pi$  is `Math.PI`,  $r$  is the radius of the cylinder's base, and  $h$  is the cylinder's height

Because the three methods are to be overloaded, they should each have the same name, but different parameter lists. Demonstrate the class in a complete program.

## 2. `InventoryItem` Class Copy Constructor

Add a copy constructor to the `InventoryItem` class. This constructor should accept an `InventoryItem` object as an argument. The constructor should assign to the `description` field the value in the argument's `description` field, and assign to the `units` field the value in the argument's `units` field. As a result, the new object will be a copy of the

argument object.



#### VideoNote The InventoryItem Class Copy Constructor Problem

### 3. Carpet Calculator

The Westfield Carpet Company has asked you to write an application that calculates the price of carpeting for rectangular rooms. To calculate the price, you multiply the area of the floor (width times length) by the price per square foot of carpet. For example, the area of floor that is 12 feet long and 10 feet wide is 120 square feet. To cover that floor with carpet that costs \$8 per square foot would cost \$960. (12 3 10 3 8 5 960.)

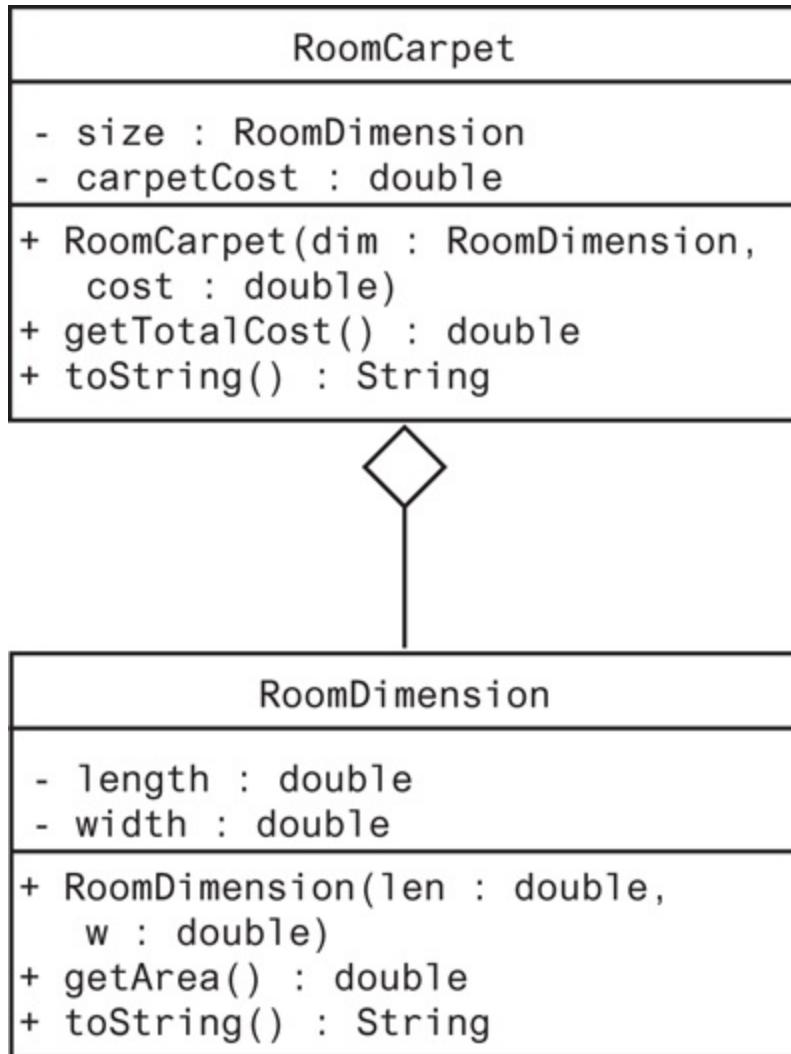
First, you should create a class named `RoomDimension` that has two fields: one for the length of the room, and one for the width. The `RoomDimension` class should have a method that returns the area of the room. (The area of the room is the room's length multiplied by the room's width.)

Next, you should create a `RoomCarpet` class that has a `RoomDimension` object as a field. It should also have a field for the cost of the carpet per square foot. The `RoomCarpet` class should have a method that returns the total cost of the carpet.

[Figure 6-21](#) is a UML diagram showing possible class designs and depicting the relationships between the classes. Once you have written these classes, use them in an application that asks the user to enter the dimensions of a room and the price per square foot of the desired carpeting. The application should display the total cost of the carpet.

## Figure 6-21 UML diagram for Programming Challenge

# 3



[Figure 6-21 Full Alternative Text](#)

## 4. LandTract Class

Make a `LandTract` class that has two fields: one for the tract's length, and one for the width. The class should have a method that returns the tract's area, as well as an `equals` method and a `toString` method.

Demonstrate the class in a program that asks the user to enter the dimensions for two tracts of land. The program should display the area of each tract of land and indicate whether the tracts are of equal size.

## 5. Month class

Write a class named `Month`. The class should have an `int` field named `monthNumber` that holds the number of the month. For example,

January would be 1, February would be 2, and so forth. In addition, provide the following methods:

- A no-arg constructor that sets the `monthNumber` field to 1.
- A constructor that accepts the number of the month as an argument. It should set the `monthNumber` field to the value passed as the argument. If a value less than 1 or greater than 12 is passed, the constructor should set `monthNumber` to 1.
- A constructor that accepts the name of the month, such as “January” or “February”, as an argument. It should set the `monthNumber` field to the correct corresponding value.
- A `setMonthNumber` method that accepts an `int` argument, which is assigned to the `monthNumber` field. If a value less than 1 or greater than 12 is passed, the method should set `monthNumber` to 1.
- A `getMonthNumber` method that returns the value in the `monthNumber` field.
- A `getMonthName` method that returns the name of the month. For example, if the `monthNumber` field contains 1, then this method should return “January”.
- A `toString` method that returns the same value as the `getMonthName` method.
- An `equals` method that accepts a `Month` object as an argument. If the argument object holds the same data as the calling object, this method should return `true`. Otherwise, it should return `false`.
- A `greaterThan` method that accepts a `Month` object as an argument. If the calling object’s `monthNumber` field is greater than the

argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.

- A `lessThan` method that accepts a `Month` object as an argument. If the calling object's `monthNumber` field is less than the argument's `monthNumber` field, this method should return `true`. Otherwise, it should return `false`.

## 6. Employee Class Modification

In Programming Challenge 1 of [Chapter 3](#), you wrote an `Employee` class. Add the following to the class:

- A constructor that accepts the following values as arguments and assigns them to the appropriate fields: employee's name, employee's ID number, department, and position.
- A constructor that accepts the following values as arguments and assigns them to the appropriate fields: employee's name and ID number. The `department` and `position` fields should be assigned an empty string ("").
- A no-arg constructor that assigns empty strings ("") to the `name`, `department`, and `position` fields, and 0 to the `idNumber` field.

Write a program that tests and demonstrates these constructors.

## 7. RetailItem Class Modification

Modify this chapter's `RetailItem` class (which uses an inner class named `CostData`) to include accessor and mutator methods for getting and setting an item's wholesale and retail cost. Demonstrate the methods in a program.

## 8. CashRegister Class

Write a `CashRegister` class that can be used with the `RetailItem` class you modified in Programming Challenge 7. The `CashRegister` class should simulate the sale of a retail item. It should have a constructor that

accepts a `RetailItem` object as an argument. The constructor should also accept an integer that represents the quantity of items being purchased. In addition, the class should have the following methods:

- The `getSubtotal` method should return the subtotal of the sale, which is the quantity multiplied by the retail cost. This method must get the retail cost from the `RetailItem` object that was passed as an argument to the constructor.
- The `getTax` method should return the amount of sales tax on the purchase. The sales tax rate is 6 percent of a retail sale.
- The `getTotal` method should return the total of the sale, which is the subtotal plus the sales tax.

Demonstrate the class in a program that asks the user for the quantity of items being purchased, and then displays the sale's subtotal, amount of sales tax, and total.

## 9. Sales Receipt File

Modify the program you wrote in Programming Challenge 8 to create a file containing a sales receipt. The program should ask the user for the quantity of items being purchased, then generate a file with contents similar to the following:

```
SALES RECEIPT
Unit Price: $10.00
Quantity: 5
Subtotal: $50.00
Sales Tax: $ 3.00
Total: $53.00
```

## 10. Parking Ticket Simulator

For this assignment, you will design a set of classes that work together to simulate a police officer issuing a parking ticket. The classes you should design are:

- The `ParkedCar` Class. This class should simulate a parked car. The

class's responsibilities are:

- – To know the car's make, model, color, license number, and the number of minutes that the car has been parked
- The `ParkingMeter` Class. This class should simulate a parking meter. The class's only responsibility is:
  - – To know the number of minutes of parking time that has been purchased
- The `ParkingTicket` Class. This class should simulate a parking ticket. The class's responsibilities are:
  - – To report the make, model, color, and license number of the illegally parked car
  - – To report the amount of the fine, which is \$25 for the first hour or part of an hour that the car is illegally parked, plus \$10 for every additional hour or part of an hour that the car is illegally parked
  - – To report the name and badge number of the police officer issuing the ticket
- The `PoliceOfficer` Class. This class should simulate a police officer inspecting parked cars. The class's responsibilities are:
  - – To know the police officer's name and badge number
  - – To examine a `ParkedCar` object and a `ParkingMeter` object, and determine whether the car's time has expired
  - – To issue a parking ticket (generate a `ParkingTicket` object) if the car's time has expired

Write a program that demonstrates how these classes collaborate.

## 11. Geometry Calculator

Design a Geometry class with the following methods:

- A static method that accepts the radius of a circle and returns the area of the circle. Use the following formula:

$$\text{Area} = \pi r^2$$

Use `Math.PI` for  $\pi$ , and the radius of the circle for  $r$ .

- A static method that accepts the length and width of a rectangle and returns the area of the rectangle. Use the following formula:

$$\text{Area} = \text{width} \times \text{length}$$

- A static method that accepts the length of a triangle's base and the triangle's height. The method should return the area of the triangle. Use the following formula:

$$\text{Area} = \text{base} \times \text{height} / 2$$

The methods should display an error message if negative values are used for the circle's radius, the rectangle's length or width, or the triangle's base or height.

Next, write a program to test the class, which displays the following menu and responds to the user's selection:

Geometry Calculator

1. Calculate the Area of a Circle
2. Calculate the Area of a Rectangle
3. Calculate the Area of a Triangle
4. Quit

Enter your choice (1-4):

Display an error message if the user enters a number outside the range of

1 through 4 when selecting an item from the menu.

## 12. Car-Instrument Simulator

For this assignment, you will design a set of classes that work together to simulate a car's fuel gauge and odometer. The classes you will design are as follows:

- The FuelGauge Class. This class simulates a fuel gauge. It's responsibilities are:
  - – To know the car's current amount of fuel, in gallons.
  - – To report the car's current amount of fuel, in gallons.
  - – To be able to increment the amount of fuel by 1 gallon. This simulates putting fuel in the car. (The car can hold a maximum of 15 gallons.)
  - – To be able to decrement the amount of fuel by 1 gallon, if the amount of fuel is greater than 0 gallons. This simulates burning fuel as the car runs.
- The odometer Class. This class simulates the car's odometer. It's responsibilities are:
  - – To know the car's current mileage.
  - – To report the car's current mileage.
  - – To be able to increment the current mileage by 1 mile. The maximum mileage the odometer can store is 999,999 miles. When this amount is exceeded, the odometer resets the current mileage to 0.
  - – To be able to work with a FuelGauge object. It should decrease the FuelGauge object's current amount of fuel by 1 gallon for every 24 miles traveled. (The car's fuel economy is 24 miles per gallon.)

Demonstrate the classes by creating instances of each. Simulate filling the car up with fuel, then run a loop that increments the odometer until the car runs out of fuel. During each loop iteration, print the car's current mileage and amount of fuel.

### 13. First to One Game

This game is meant for two or more players. In the game, each player starts out with 50 points, as each player takes a turn rolling a pair of dice; the amount generated by the dice is subtracted from the player's points. The first player with exactly one point remaining wins. If a player's remaining points minus the amount generated by the dice results in a value less than one, then the amount should be added to the player's points. (As an alternative, the game can be played with a set number turns. In this case, the player with the amount of points closest to one, when all rounds have been played, wins.)

Write a program that simulates the game being played by two players. Use the `Die` class that was presented in [Chapter 4](#) to simulate the dice. Write a `Player` class to simulate the players.

### 14. Heads or Tails Game

This game is meant for two or more players. In this game, the players take turns flipping a coin. Before the coin is flipped, players should guess if the coin will land heads up or tails up. If a player guesses correctly, then that player is awarded a point. If a player guesses incorrectly, then that player will lose a point. The first player to score five points is the winner.

Write a program that simulates the game being played by two players. Use the `Coin` class that you wrote as an assignment in [Chapter 5](#) (Programming Challenge 20) to simulate the coin. Write a `Player` class to simulate the players.

# Chapter 7 Arrays and the ArrayList Class

## Topics

1. [7.1 Introduction to Arrays](#)
2. [7.2 Processing Array Contents](#)
3. [7.3 Passing Arrays as Arguments to Methods](#)
4. [7.4 Some Useful Array Algorithms and Operations](#)
5. [7.5 Returning Arrays from Methods](#)
6. [7.6 String Arrays](#)
7. [7.7 Arrays of Objects](#)
8. [7.8 The Sequential Search Algorithm](#)
9. [7.9 The Selection Sort and the Binary Search Algorithms](#)
10. [7.10 Two-Dimensional Arrays](#)
11. [7.11 Arrays with Three or More Dimensions](#)
12. [7.12 Command-Line Arguments and Variable-Length Argument Lists](#)
13. [7.13 The ArrayList Class](#)
14. [7.14 Common Errors to Avoid](#)

# 7.1 Introduction to Arrays

## Concept:

An array can hold multiple values of the same data type simultaneously.

The primitive variables you have worked with so far are designed to hold only one value at a time. Each of the variable declarations in [Figure 7-1](#) causes only enough memory to be reserved to hold one value of the specified data type.

## Figure 7-1 Variable declarations and their memory allocations

`int count;`      Enough memory to hold one int.

1234

`double number;`      Enough memory to hold one double.

1234.55

`char letter;`      Enough memory to hold one char.

A

## [Figure 7-1 Full Alternative Text](#)

An array, however, is an object that can store a group of values, all of the same type. For example, suppose a weather-related application records the high temperature each day for a week. It would record a total of seven values, one for each day. All of these values would be doubles. Such an application could store the values in an array of seven doubles. Or, suppose a sales-related application records the number of items sold each month for a year. It would record a total of 12 values, one for each month. All of these values would be ints. That application could store the values in an array of 12 ints.

Creating and using an array in Java is similar to creating and using any other type of object: You declare a reference variable, and use the new key word to create an instance of the array in memory. Here is an example of a statement that declares an array reference variable:

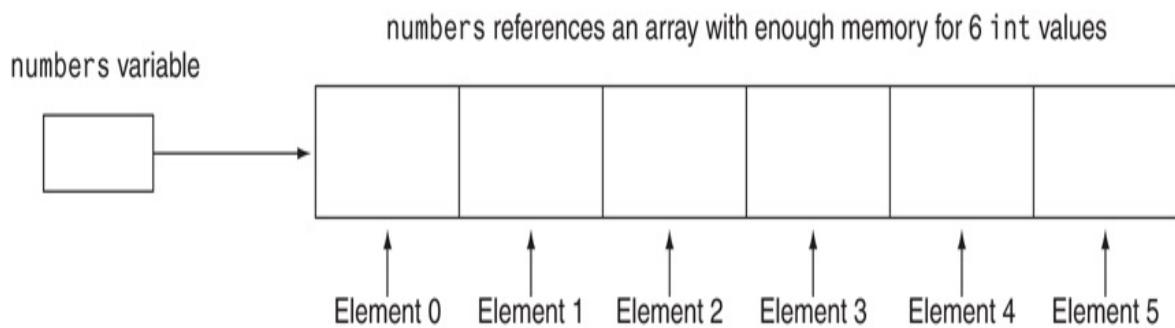
```
int[] numbers;
```

This statement declares numbers as an array reference variable. The numbers variable can reference an array of int values. Notice this statement looks like a regular int variable declaration except for the set of brackets that appear after the word int. The brackets indicate that this variable is a reference to an int array. Declaring an array reference variable does not create an array. The next step in the process is to use the new key word to create an array and assign its address to the numbers variable. The following statement shows an example:

```
numbers = new int[6];
```

The number inside the brackets is the array's *size declarator*. It indicates the number of *elements*, or values, the array can hold. When this statement is executed, numbers will reference an array that can hold six elements, each one an int. This is depicted in [Figure 7-2](#).

## Figure 7-2 The numbers array



[Figure 7-2 Full Alternative Text](#)

As with any other type of object, it is possible to declare a reference variable and create an instance of an array with one statement. Here is an example:

```
int[] numbers = new int[6];
```

Arrays of any data type can be declared. The following are all valid array declarations:

```
float[] temperatures = new float[100];
char[] letters = new char[41];
long[] units = new long[50];
double[] sizes = new double[1200];
```

An array's size declarator must be a nonnegative integer expression. It can be either a literal value, as shown in the previous examples, or a variable. It is a common practice to use a `final` variable as a size declarator. Here is an example:

```
final int ARRAY_SIZE = 6;
int[] numbers = new int[ARRAY_SIZE];
```

This practice makes programs easier to maintain. As you will see, programs that use an array often refer to its size. When we store the size of an array in a variable, we can use the variable instead of a literal number when we refer to the size of the array. If we ever need to change the array's size, we need only to change the value of the variable. The variable should be `final` so its contents cannot be changed during the program's execution.



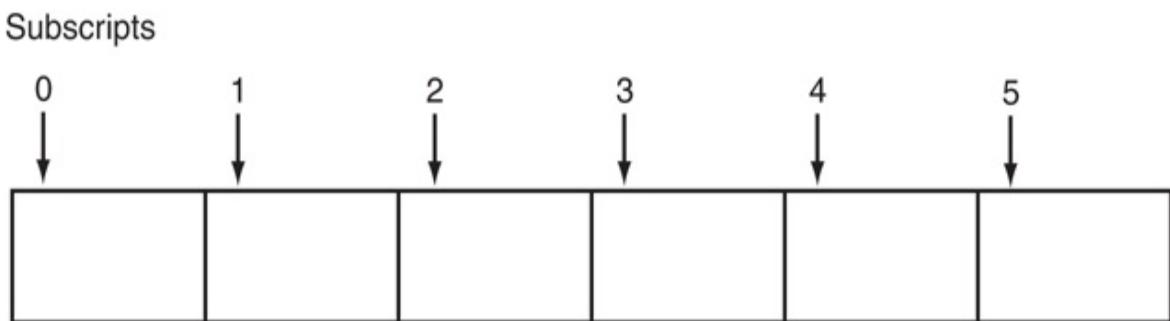
## Note:

Once an array is created, its size cannot be changed.

# Accessing Array Elements

Although an array has only one name, the elements in the array may be accessed and used as individual variables. This is possible because each element is assigned a number known as a *subscript*. A subscript is used as an index to pinpoint a specific element within an array. The first element is assigned the subscript 0, the second element is assigned 1, and so forth. The six elements in the `numbers` array (described earlier) would have the subscripts 0 through 5. This is shown in [Figure 7-3](#).

**Figure 7-3 Subscripts for the numbers array**



The `numbers` array has six elements, numbered 0 through 5.

Subscript numbering always starts at 0. The subscript of the last element in an array is one less than the total number of elements in the array. This means that for the `numbers` array, which has six elements, 5 is the subscript for the last element.

Each element in the `numbers` array, when accessed by its subscript, can be used as an `int` variable. Here is an example of a statement that stores the number 20 in the first element of the array:

```
numbers[0] = 20;
```

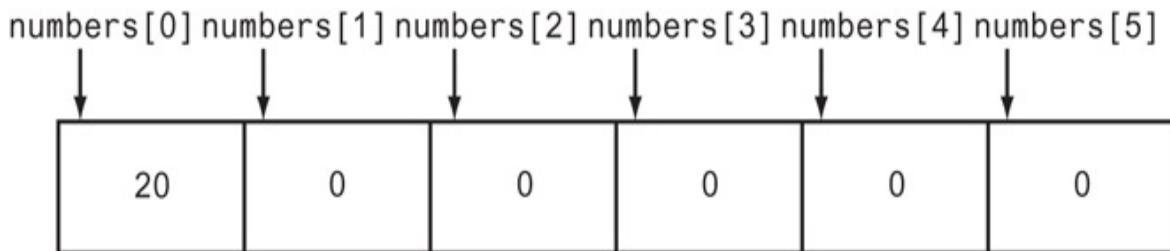


## Note:

The expression `numbers[0]` is pronounced “numbers sub zero.” You would read this assignment statement as “numbers sub zero is assigned twenty.”

[Figure 7-4](#) illustrates the contents of the `numbers` array after the previously shown statement assigns 20 to `numbers[0]`.

## Figure 7-4 Contents of the array after 20 is assigned to numbers[0]



[Figure 7-4 Full Alternative Text](#)



## Note:

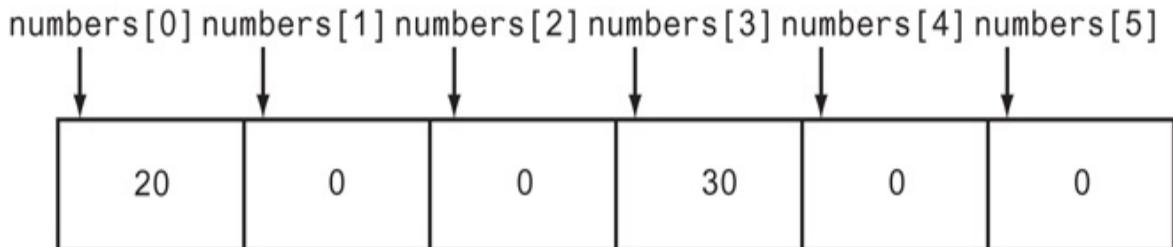
By default, Java initializes array elements with 0. In [Figure 7-4](#), values have not been stored in elements 1 through 5, so they are shown as 0s.

The following statement stores the integer 30 in `numbers[3]`, which is the fourth element of the `numbers` array:

```
numbers[3] = 30;
```

[Figure 7-5](#) illustrates the contents of the array after this statement executes.

## Figure 7-5 Contents of the array after 30 is assigned to numbers[3]



[Figure 7-5 Full Alternative Text](#)

By this point, you should understand the difference between the array size declarator and a subscript. The number inside the brackets in a statement that uses the new keyword to create an array is the size declarator. It indicates the number of elements that the array has. The number inside the brackets in an assignment statement or any statement that works with the contents of an array is a subscript. It is used to access a specific element in the array.

## Inputting and Outputting Array Contents

If you want to input values into an array, you must input the values one at a time into the individual array elements. For example, if you have an array with five elements, then inputting values into the array will require five input

operations, one for each element. The same is true for outputting the contents of an array. If you want to display the contents of an array with five elements, then you must display the contents of each individual array element.

[Code Listing 7-1](#) gives an example of inputting and outputting an array's contents. Values are read from the keyboard and stored in an array. The value of each element is then displayed.

## Code Listing 7-1 (ArrayDemo1.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program shows values being read into an array's
5 * elements and then displayed.
6 */
7
8 public class ArrayDemo1
9 {
10 public static void main(String[] args)
11 {
12 final int NUM_EMPLOYEES = 3; // Number of employees
13
14 // Create an array to hold employee hours.
15 int[] hours = new int[NUM_EMPLOYEES];
16
17 // Create a Scanner object for keyboard input.
18 Scanner keyboard = new Scanner(System.in);
19
20 System.out.println("Enter the hours worked by " +
21 NUM_EMPLOYEES + " employees.");
22
23 // Get employee 1's hours.
24 System.out.print("Employee 1: ");
25 hours[0] = keyboard.nextInt();
26
27 // Get employee 2's hours.
28 System.out.print("Employee 2: ");
29 hours[1] = keyboard.nextInt();
30
31 // Get employee 3's hours.
32 System.out.print("Employee 3: ");
33 hours[2] = keyboard.nextInt();
```

```
34
35 // Display the values in the array.
36 System.out.println("The hours you entered are:");
37 System.out.println(hours[0]);
38 System.out.println(hours[1]);
39 System.out.println(hours[2]);
40 }
41 }
```

## Program Output with Example Input

Enter the hours worked by 3 employees.

Employee 1: **40** 

Employee 2: **20** 

Employee 3: **15** 

The hours you entered are:

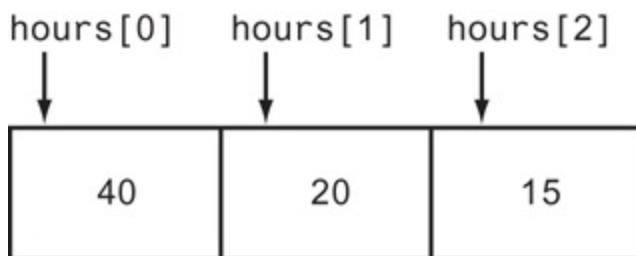
40

20

15

[Figure 7-6](#) shows the contents of the hours array with the values entered by the user in the example output.

## Figure 7-6 Contents of the hours array



[VideoNote](#) Accessing Array Elements in a Loop

Subscript numbers can be stored in variables. This makes it possible to use a

loop to “cycle through” or “step through” an entire array, performing the same operation on each element. For example, [Code Listing 7-1](#) could be simplified by using two `for` loops: one for inputting the values into the array, and the other for displaying the contents of the array. This is shown in [Code Listing 7-2](#).

## Code Listing 7-2 (ArrayDemo2.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program shows an array being processed with loops.
5 */
6
7 public class ArrayDemo2
8 {
9 public static void main(String[] args)
10 {
11 final int NUM_EMPLOYEES = 3; // Number of employees
12
13 // Create an array to hold employee hours.
14 int[] hours = new int[NUM_EMPLOYEES];
15
16 // Create a Scanner object for keyboard input.
17 Scanner keyboard = new Scanner(System.in);
18
19 System.out.println("Enter the hours worked by " +
20 NUM_EMPLOYEES + " employees.");
21
22 // Cycle through the array, getting each
23 // employee's hours.
24 for (int index = 0; index < NUM_EMPLOYEES; index++)
25 {
26 System.out.print("Employee " + (index + 1) + ": ");
27 hours[index] = keyboard.nextInt();
28 }
29
30 // Cycle through the array displaying each element.
31 System.out.println("The hours you entered are:");
32 for (int index = 0; index < NUM_EMPLOYEES; index++)
33 System.out.println(hours[index]);
34 }
35 }
```

## Program Output with Example Input Shown in Bold

Enter the hours worked by 3 employees.

Employee 1: **40**

Employee 2: **20**

Employee 3: **15**

The hours you entered are:

40

20

15

Let's take a closer look at the first loop in this program, which appears in lines 24 through 28. Notice that the loop's control variable, `index`, is used as a subscript in line 27:

```
hours[index] = keyboard.nextInt();
```

The variable `index` starts at 0. During the loop's first iteration, the user's input is stored in `hours[0]`. Then, `index` is incremented, so its value becomes 1. During the next iteration, the user's input is stored in `hours[1]`. This continues until values have been stored in all of the elements of the array. Notice that the loop correctly starts and ends the control variable with valid subscript values (0 through 2), as illustrated in [Figure 7-7](#). This ensures that only valid subscripts are used.

## Figure 7-7 Annotated loop

The variable `index` starts at 0, which is the first valid subscript value.

The loop ends before the variable `index` reaches 3, which is the first invalid subscript value.

```
for (int index = 0; index < NUM_EMPLOYEES; index++)
{
 System.out.print("Employee " + (index + 1) + ": ");
 hours[index] = keyboard.nextInt();
}
```

[Figure 7-7 Full Alternative Text](#)

# Java Performs Bounds Checking

Java performs array bounds checking, which means that it does not allow a statement to use a subscript outside the range of valid subscripts for an array. For example, the following code creates an array with 10 elements:

```
final int ARRAY_SIZE = 10;
int[] values = new int[ARRAY_SIZE];
```

The valid subscripts for the array are 0 through 9. Java will not allow a statement to use a subscript less than 0 or greater than 9 with this array. Bounds checking occurs at runtime. The Java compiler does not display an error message when it processes a statement that uses an invalid subscript. Instead, when the statement executes, the program throws an exception and terminates. (Exceptions will be discussed in [Chapter 10](#).) For instance, the program in [Code Listing 7-3](#) declares a three-element array, but attempts to store four values in the array.

## Code Listing 7-3 (**InvalidSubscript.java**)

```
1 /**
2 * This program uses an invalid subscript with an array.
3 */
4
5 public class InvalidSubscript
6 {
7 public static void main(String[] args)
8 {
9 // Create an array with three elements.
10 int[] values = new int[3];
11
12 System.out.println("I will attempt to store four " +
13 "numbers in a 3-element array.");
14
15 for (int index = 0; index < 4; index++)
```

```
16 {
17 System.out.println("Now processing element " + index);
18 values[index] = 10;
19 }
20 }
21 }
```

## Program Output

```
I will attempt to store four numbers in a 3-element array.
Now processing element 0
Now processing element 1
Now processing element 2
Now processing element 3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
 at InvalidSubscript.main(InvalidSubscript.java:18)
```

When the program attempted to store a value in `values[3]`, it halted and an error message was displayed.



## Note:

The error message you see may be different, depending on your system.

# Watch Out for Off-by-One Errors

Because array subscripts start at 0 rather than 1, you have to be careful not to perform an *off-by-one error*. For example, look at the following code:

```
// This code has an off-by-one error.
final int ARRAY_SIZE = 100;
int[] numbers = new int[ARRAY_SIZE];
for (int index = 1; index <= ARRAY_SIZE; index++)
 numbers[index] = 99;
```

The intent of this code is to create an array of integers with 100 elements, and store the value 99 in each element. However, this code has an off-by-one error. The loop uses its control variable, `index`, as a subscript with the

numbers array. During the loop's execution, the variable index takes on the values 1 through 100, when it should take on the values 0 through 99. As a result, the first element, which is at subscript 0, is skipped. In addition, the loop attempts to use 100 as a subscript during the last iteration. Because 100 is an invalid subscript, the program will throw an exception and halt.

## Array Initialization

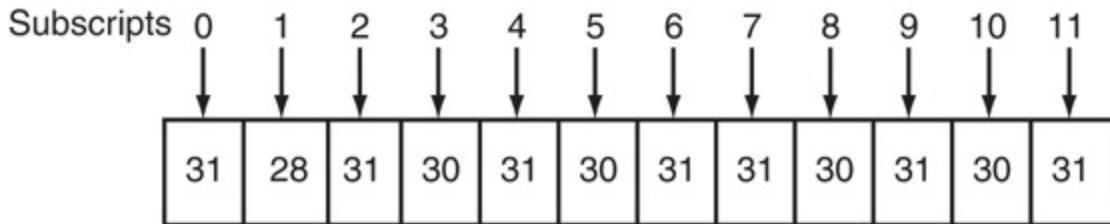
When you create an array, Java allows you to initialize its elements with values. Here is an example:

```
int[] days = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

This statement declares the reference variable `days`, creates an array in memory, and stores initial values in the array's elements. The series of values inside the braces and separated with commas is called an *initialization list*. These values are stored in the array elements in the order they appear in the list. (The first value, 31, is stored in `days[0]`, the second value, 28, is stored in `days[1]`, and so forth.) Note you do not use the `new` key word when you use an initialization list. Java automatically creates the array and stores the values in the initialization list in it.

The Java compiler determines the size of the array by the number of items in the initialization list. Because there are 12 items in the example statement's initialization list, the array will have 12 elements. [Figure 7-8](#) shows the contents of the array after the initialization. The program in [Code Listing 7-4](#) demonstrates the array.

## Figure 7-8 The contents of the array after the initialization



[Figure 7-8 Full Alternative Text](#)

## Code Listing 7-4 (ArrayInitialization.java)

```

1 /**
2 * This program shows an array being initialized.
3 */
4
5 public class ArrayInitialization
6 {
7 public static void main(String[] args)
8 {
9 final int MONTHS = 12; // Number of months
10
11 // Create and initialize an array.
12 int[] days = { 31, 28, 31, 30, 31, 30,
13 31, 31, 30, 31, 30, 31 };
14
15 // Display the days in each month.
16 for (int index = 0; index < MONTHS; index++)
17 {
18 System.out.println("Month " + (index + 1) +
19 " has " + days[index] + " days.");
20 }
21 }
22 }
```

### Program Output

```

Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
```

```
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

Java allows you to spread the initialization list across multiple lines. Both of the following array declarations are equivalent:

```
double[] coins = { 0.05, 0.1, 0.25, 0.5, 1.0 };
double[] coins = { 0.05,
 0.1,
 0.25,
 0.5,
 1.0 };
```

## Alternate Array Declaration Notation

Java allows you to use two different styles when declaring array reference variables. The first style is the one that we have used in this book, with the brackets immediately following the data type, as shown here:

```
int[] numbers;
```

In the second style, the brackets are placed after the variable name, as shown here:

```
int numbers[];
```

Both of these statements accomplish the same thing: They declare that `numbers` is a reference to an `int` array. The difference between the two styles is noticed when more than one variable is declared in the same statement. For example, look at the following statement:

```
int[] numbers, codes, scores;
```

This statement declares three variables: `numbers`, `codes`, and `scores`. All

three are references to `int` arrays. This makes perfect sense because `int[]` is the data type for all the variables declared in the statement. Now look at the following statement, which uses the alternate notation:

```
int numbers[], codes, scores;
```

This statement declares the same three variables, but only `numbers` is a reference to an `int` array. The `codes` and `scores` variables are regular `int` variables. This is because `int` is the data type for all the variables declared in the statement, and only `numbers` is followed by the brackets. To declare all three of these variables as references to `int` arrays using the alternate notation, you need to write a set of brackets after each variable name. Here is an example:

```
int numbers[], codes[], scores[];
```

The first style is the standard notation for most Java programmers, so we will continue to use that style in this book.



## Checkpoint

1. 7.1 Write statements that create the following arrays:
  1. A 100-element `int` array referenced by the variable `employeeNumbers`
  2. A 25-element `double` array referenced by the variable `payRates`
  3. A 14-element `float` array referenced by the variable `miles`
  4. A 1000-element `char` array referenced by the variable `letters`
2. 7.2 What's wrong with the following array declarations?

```
int[] readings = new int[-1];
double[] measurements = new double[4.5];
```

3. 7.3 What would the valid subscript values be in a four-element array of

doubles?

4. 7.4 What is the difference between an array's size declarator and a subscript?
5. 7.5 What does it mean for a subscript to be out-of-bounds?
6. 7.6 What happens in Java when a program tries to use a subscript that is out-of-bounds?
7. 7.7 What is the output of the following code?

```
int[] values = new int[5];

for (int count = 0; count < 5; count++)
 values[count] = count + 1;

for (int count = 0; count < 5; count++)
 System.out.println(values[count]);
```

8. 7.8 Write a statement that creates and initializes a double array with the following values: 1.7, 6.4, 8.9, 3.1, and 9.2. How many elements are there in the array?

## 7.2 Processing Array Contents

### Concept:

Individual array elements are processed like any other type of variable.

Working with an individual array element is no different than working with a variable. For example, the following statement multiplies `hours[3]` by the variable `payRate`. The result is stored in the variable `grossPay`.

```
grossPay = hours[3] * payRate;
```

The following are examples of preincrement and postincrement operations on array elements:

```
int[] score = {7, 8, 9, 10, 11};
++score[2]; // Preincrement operation
score[4]++; // Postincrement operation
```

When using increment and decrement operators, be careful not to use the operator on the subscript when you intend to use it on the array element. For example, the following statement decrements the variable `count`, but does nothing to `amount[count]`:

```
amount[count--];
```

[Code Listing 7-5](#) demonstrates the use of array elements in a simple mathematical statement. A loop steps through each element of the array, using the elements to calculate the gross pay of five employees.

### Code Listing 7-5 (PayArray.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program stores in an array the hours worked by
5 * five employees who all make the same hourly wage.
6 */
7
8 public class PayArray
9 {
10 public static void main(String[] args)
11 {
12 final int NUM_EMPLOYEES = 5; // Number of employees
13 double payRate, // Hourly pay rate
14 grossPay; // Gross pay
15
16 // Create an array for employee hours.
17 int[] hours = new int[NUM_EMPLOYEES];
18
19 // Create a Scanner object for keyboard input.
20 Scanner keyboard = new Scanner(System.in);
21
22 System.out.println("Enter the hours worked by " +
23 NUM_EMPLOYEES + " employees who " +
24 "all earn the same hourly rate.");
25
26 // Get each employee's hours worked.
27 for (int index = 0; index < NUM_EMPLOYEES; index++)
28 {
29 System.out.print("Employee #" + (index + 1) + ": ");
30 hours[index] = keyboard.nextInt();
31 }
32
33 // Get the hourly pay rate.
34 System.out.print("Enter each employee's hourly rate: ");
35 payRate = keyboard.nextDouble();
36
37 // Display each employee's gross pay.
38 System.out.println("Gross pay for each employee:");
39 for (int index = 0; index < NUM_EMPLOYEES; index++)
40 {
41 grossPay = hours[index] * payRate;
42 System.out.printf("Employee #%d: $%, .2f\n",
43 (index + 1), grossPay);
44 }
45 }
46 }
```

## Program Output with Example Input Shown in Bold

Enter the hours worked by 5 employees who all earn the same hourly rate:

Employee #1: **10**   
Employee #2: **20**   
Employee #3: **30**   
Employee #4: **40**   
Employee #5: **50**

Enter each employee's hourly rate: **10**

Gross pay for each employee:

Employee #1: \$100.00  
Employee #2: \$200.00  
Employee #3: \$300.00  
Employee #4: \$400.00  
Employee #5: \$500.00

In line 41 of the program, the following statement assigns the value of `hours[index]` times `payRate` to the `grossPay` variable:

```
grossPay = hours[index] * payRate;
```

Array elements may also be used in relational expressions. For example, the following `if` statement determines whether `cost[20]` is less than `cost[0]`:

```
if (cost[20] < cost[0])
```

And the following `while` loop iterates as long as `value[count]` does not equal 0:

```
while (value[count] != 0)
{
 Statements
}
```

[Code Listing 7-6](#), a modification of [Code Listing 7-5](#), includes overtime wages in the gross pay. If an employee works more than 40 hours, an overtime pay rate of 1.5 times the regular pay rate is used for the excess hours.

# Code Listing 7-6 (Overtime.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program stores in an array the hours worked by
5 * five employees who all make the same hourly wage.
6 * Overtime wages are paid for hours greater than 40.
7 */
8
9 public class Overtime
10 {
11 public static void main(String[] args)
12 {
13 final int NUM_EMPLOYEES = 5; // Number of employees
14 double payRate, // Hourly pay rate
15 grossPay, // Gross pay
16 overtime; // Overtime wages
17
18 // Create an array for employee hours.
19 int[] hours = new int[NUM_EMPLOYEES];
20
21 // Create a Scanner object for keyboard input.
22 Scanner keyboard = new Scanner(System.in);
23
24 System.out.println("Enter the hours worked by " +
25 NUM_EMPLOYEES + " employees who " +
26 "all earn the same hourly rate.");
27
28 // Get each employee's hours worked.
29 for (int index = 0; index < NUM_EMPLOYEES; index++)
30 {
31 System.out.print("Employee #" + (index + 1) + ": ");
32 hours[index] = keyboard.nextInt();
33 }
34
35 // Get the hourly pay rate.
36 System.out.print("Enter the hourly rate for each employee:
37 payRate = keyboard.nextDouble();
38
39 // Display each employee's gross pay.
40 System.out.println("Here is the gross pay for each employee
41 for (int index = 0; index < NUM_EMPLOYEES; index++)
42 {
43 if (hours[index] > 40)
```

```

44 {
45 // Calculate base pay
46 grossPay = 40 * payRate;
47
48 // Calculate overtime pay
49 overtime = (hours[index] - 40) * (1.5 * payRate);
50
51 // Add base pay and overtime pay
52 grossPay += overtime;
53 }
54 else
55 grossPay = hours[index] * payRate;
56
57 System.out.printf("Employee #%-d: $%, .2f\n",
58 (index + 1), grossPay);
59 }
60 }
61 }
```

## Program Output with Example Input Shown in Bold

Enter the hours worked by 5 employees who all earn the same hourly rate:

Employee #1: **10**

Employee #2: **40**

Employee #3: **60**

Employee #4: **50**

Employee #5: **30**

Enter the hourly rate for each employee: **10**

Here is the gross pay for each employee:

Employee #1: \$100.00

Employee #2: \$400.00

Employee #3: \$700.00

Employee #4: \$550.00

Employee #5: \$300.00

As the second `for` loop in [Code Listing 7-6](#) is stepping through the array, it tests each element with the following `if` statement in line 49:

```
if (hours[index] > 40)
```

If the array element is greater than 40, an overtime formula is used to calculate the employee's gross pay.

# Array Length

Each array in Java has a public field named `length`. This field contains the number of elements in the array. For example, consider an array created by the following statement:

```
double[] temperatures = new double[25];
```

Because the `temperatures` array has 25 elements, the following statement would assign 25 to the variable `size`.

```
size = temperatures.length;
```

The `length` field can be useful when processing the entire contents of an array. For example, the following loop steps through an array and displays the contents of each element. The array's `length` field is used in the test expression as the upper limit for the loop control variable.

```
for (int index = 0; index < temperatures.length; index++)
 System.out.println(temperatures[index]);
```



## Warning!

Be careful not to cause an off-by-one error when using the `length` field as the upper limit of a subscript. The `length` field contains the number of elements that an array has. The largest subscript that an array has is `length - 1`.



## Note:

You cannot change the value of an array's `length` field.

# The Enhanced for Loop

Java provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing. It is known as the *enhanced for loop*. Here is the general format of the enhanced `for` loop:

```
for (dataType elementVariable : array)
 statement;
```

The enhanced `for` loop is designed to iterate once for every element in an array. Each time the loop iterates, it copies an array element to a variable. Let's look at the syntax more closely:

- *dataType elementVariable* is a variable declaration. This variable will receive the value of a different array element during each loop iteration. During the first loop iteration, it receives the value of the first element; during the second iteration, it receives the value of the second element; and so on. This variable must be of the same data type as the array elements, or a type that the elements can automatically be converted to.
- *array* is the name of an array that you wish the loop to operate on. The loop will iterate once for every element in the array.
- *statement* is a statement that executes during a loop iteration.

For example, assume that we have the following array declaration:

```
int[] numbers = { 3, 6, 9 };
```

We can use the following enhanced `for` loop to display the contents of the `numbers` array:

```
for (int val : numbers)
 System.out.println(val);
```

Because the `numbers` array has three elements, this loop will iterate three times. The first time it iterates, the `val` variable will receive the value in `numbers[0]`. During the second iteration, `val` will receive the value in `numbers[1]`. During the third iteration, `val` will receive the value in `numbers[2]`. The code's output will be as follows:

If you need to execute more than one statement in the enhanced `for` loop, simply enclose the block of statements in a set of braces. Here is an example:

```
int[] numbers = { 3, 6, 9 };
for (int val : numbers)
{
 System.out.print("The next value is ");
 System.out.println(val);
}
```

This code will produce the following output:

```
The next value is 3
The next value is 6
The next value is 9
```

## **The Enhanced for Loop versus the Traditional for Loop**

When you need to access the values stored in an array, from the first element to the last element, the enhanced `for` loop is simpler to use than the traditional `for` loop. With the enhanced `for` loop, you do not have to be concerned about the size of the array, and you do not have to create an “index” variable to hold subscripts. However, there are circumstances in which the enhanced `for` loop is not adequate. You cannot use the enhanced `for` loop if:

- you need to change the contents of an array element.
- you need to work through the array elements in reverse order.
- you need to access some of the array elements, but not all of them.
- you need to simultaneously work with two or more arrays within the loop.

- you need to refer to the subscript number of a particular element.

In any of these circumstances, you should use the traditional `for` loop to process the array.

## Letting the User Specify an Array's Size

Java allows you to use an integer variable to specify an array's size declarator. This makes it possible to allow the user to specify an array's size. The following code shows an example. (Assume that `keyboard` references a `Scanner` object.)

```
int size;
int[] numbers;
System.out.print("How many numbers do you have? ");
size = keyboard.nextInt();
numbers = new int[size];
```

[Code Listing 7-7](#) demonstrates this, as well as the use of the `length` field. It stores a number of test scores in an array then displays them.

## Code Listing 7-7 (`DisplayTestScores.java`)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates how the user may specify an
5 * array's size.
6 */
7
8 public class DisplayTestScores
9 {
10 public static void main(String[] args)
11 {
12 int numTests; // Number of tests
```

```

13 int[] tests; // To reference an array of scores
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the number of test scores.
19 System.out.print("How many tests do you have? ");
20 numTests = keyboard.nextInt();
21
22 // Create an array to hold that number of scores.
23 tests = new int[numTests];
24
25 // Get the individual test scores.
26 for (int index = 0; index < tests.length; index++)
27 {
28 System.out.print("Enter test score " +
29 (index + 1) + ": ");
30 tests[index] = keyboard.nextInt();
31 }
32
33 // Display the test scores.
34 System.out.println();
35 System.out.println("Here are the scores you entered:");
36 for (int index = 0; index < tests.length; index++)
37 System.out.println(tests[index]);
38 }
39 }
```

## Program Output with Example Input Shown in Bold

```

How many tests do you have? 5 
Enter test score 1: 72 
Enter test score 2: 85 
Enter test score 3: 81 
Enter test score 4: 94 
Enter test score 5: 99 
Here are the scores you entered:
72 85 81 94 99
```

This program allows the user to determine the size of the array. The statement in line 23 creates the array, using the numTests variable to determine its size. The program then uses two for loops. The first, in lines 26 through 31, allows the user to input each test score, and the second, in lines 36 and 37,

displays all of the test scores. Both loops use the `length` member to control their number of iterations.

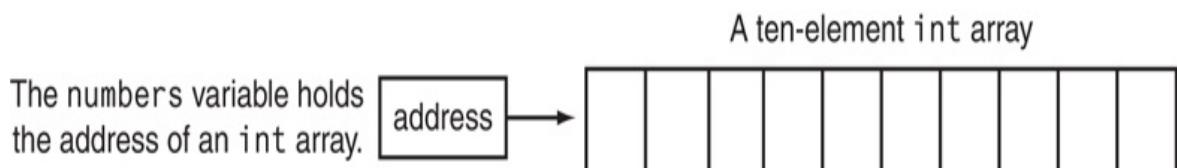
# Reassigning Array Reference Variables

It is possible to reassign an array reference variable to a different array, as demonstrated by the following code.

```
// Create an array referenced by the numbers variable.
int[] numbers = new int[10];
// Reassign numbers to a new array.
numbers = new int[5];
```

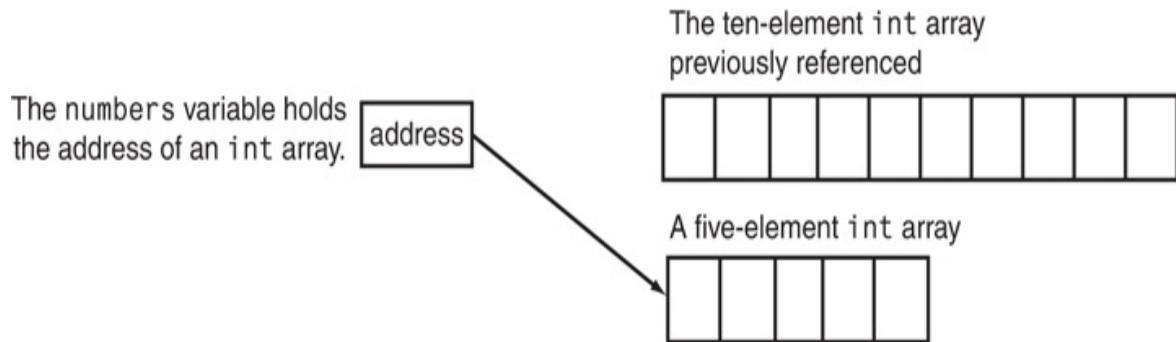
The first statement creates a ten-element integer array and assigns its address to the `numbers` variable. This is illustrated in [Figure 7-9](#).

**Figure 7-9 The `numbers` variable references a ten-element array**



The second statement then creates a five-element integer array and assigns its address to the `numbers` variable. The address of the five-element array takes the place of the address of the ten-element array. After this statement executes, the `numbers` variable references the five-element array instead of the ten-element array. This is illustrated in [Figure 7-10](#).

## Figure 7-10 The numbers variable references a five-element array



Because the ten-element array is no longer referenced, it becomes a candidate for garbage collection. The next time the garbage collector runs, the array will be deleted.

## Copying Arrays

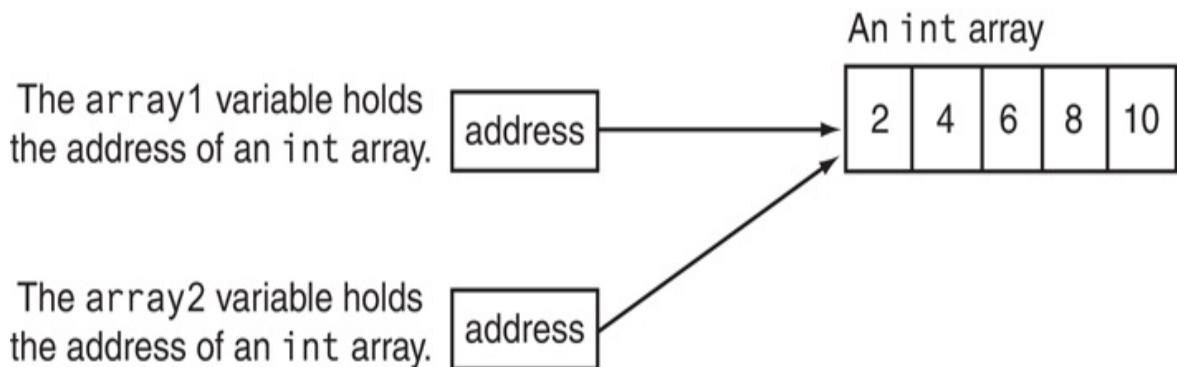
Because an array is an object, there is a distinction between an array and the variable that references it. The array and the reference variable are two separate entities. This is important to remember when you wish to copy the contents of one array to another. You might be tempted to write something like the following code, thinking that you are copying an array.

```
int[] array1 = { 2, 4, 6, 8, 10 };
int[] array2 = array1; // This does not copy array1.
```

The first statement creates an array and assigns its address to the `array1` variable. The second statement assigns `array1` to `array2`. This does not make a copy of the array referenced by `array1`. Rather, it makes a copy of the address stored in `array1` and stores it in `array2`. After this statement executes, both the `array1` and `array2` variables will reference the same array. Recall from [Chapter 6](#) that this is called a reference copy. This is illustrated

in [Figure 7-11](#).

## Figure 7-11 Both array1 and array2 reference the same array



[Code Listing 7-8](#) demonstrates the assigning of an array's address to two reference variables. Regardless of which variable the program uses, it is working with the same array.

## Code Listing 7-8 (SameArray.java)

```
1 /**
2 * This program demonstrates that two variables can
3 * reference the same array.
4 */
5
6 public class SameArray
7 {
8 public static void main(String[] args)
9 {
10 int[] array1 = { 2, 4, 6, 8, 10 };
11 int[] array2 = array1;
12
13 // Change one of the elements using array1.
14 array1[0] = 200;
15 }
```

```
16 // Change one of the elements using array2.
17 array2[4] = 1000;
18
19 // Display all the elements using array1
20 System.out.println("The contents of array1:");
21 for (int value : array1)
22 System.out.print(value + " ");
23 System.out.println();
24
25 // Display all the elements using array2
26 System.out.println("The contents of array2:");
27 for (int value : array2)
28 System.out.print(value + " ");
29 System.out.println();
30 }
31 }
```

## Program Output

The contents of array1:

200 4 6 8 1000

The contents of array2:

200 4 6 8 1000

The program in [Code Listing 7-8](#) illustrates that you cannot copy an array by merely assigning one array reference variable to another. Instead, you must copy the individual elements of one array to another. Usually, this is best done with a loop, such as:

```
final int ARRAY_SIZE = 5;
int[] firstArray = {5, 10, 15, 20, 25};
int[] secondArray = new int[ARRAY_SIZE];

for (int index = 0; index < firstArray.length; index++)
 secondArray[index] = firstArray[index];
```

The loop in this code copies each element of `firstArray` to the corresponding element of `secondArray`. This is demonstrated by [Code Listing 7-9](#), which is a modification of the program in [Code Listing 7-8](#). This version of the program makes a copy of the array referenced by `array1`.

# Code Listing 7-9 (CopyArray.java)

```
1 /**
2 * This program demonstrates how to copy an array.
3 */
4
5 public class CopyArray
6 {
7 public static void main(String[] args)
8 {
9 final int ARRAY_SIZE = 5; // Sizes of the arrays.
10 int[] array1 = { 2, 4, 6, 8, 10 };
11 int[] array2 = new int[ARRAY_SIZE];
12
13 // Make array 2 reference a copy of array1.
14 for (int index = 0; index < array1.length; index++)
15 array2[index] = array1[index];
16
17 // Change one of the elements of array1.
18 array1[0] = 200;
19
20 // Change one of the elements of array2.
21 array2[4] = 1000;
22
23 // Display all the elements using array1
24 System.out.println("The contents of array1:");
25 for (int value : array1)
26 System.out.print(value + " ");
27 System.out.println();
28
29 // Display all the elements using array2
30 System.out.println("The contents of array2:");
31 for (int value : array2)
32 System.out.print(value + " ");
33 System.out.println();
34 }
35 }
```

## Program Output

The contents of array1:

200 4 6 8 10

The contents of array2:

2 4 6 8 1000



# Checkpoint

1. 7.9 Look at the following statements.

```
int[] numbers1 = { 1, 3, 6, 9 };
int[] numbers2 = { 2, 4, 6, 8 };
int result;
```

Write a statement that multiplies element 0 of the `numbers1` array by element 3 of the `numbers2` array and assigns the result to the `result` variable.

2. 7.10 A program uses a variable named `array` that references an array of integers. You do not know the number of elements in the array. Write a `for` loop that stores -1 in each element of the array.
3. 7.11 A program has the following declaration:

```
double[] values;
```

Write code that asks the user for the size of the array, then creates an array of the specified size, referenced by the `values` variable.

4. 7.12 Look at the following statements:

```
final int ARRAY_SIZE = 7;
int[] a = { 1, 2, 3, 4, 5, 6, 7 };
int[] b = new int[ARRAY_SIZE];
```

Write code that copies the `a` array to the `b` array. The code must perform a deep copy.

# 7.3 Passing Arrays as Arguments to Methods

## Concept:

An array can be passed as an argument to a method. To pass an array, you pass the value in the variable that references the array.



**VideoNote** Passing an Array to a Method

Quite often, you'll want to write methods that process the data in arrays. As you will see, methods can be written to store values in an array, display an array's contents, total all of an array's elements, calculate their average, and so forth. Usually, such methods accept an array as an argument.

When a single element of an array is passed to a method, it is handled like any other variable. For example, [Code Listing 7-10](#) shows a loop that passes each element of the array numbers to the method `showValue`.

## Code Listing 7-10 (`PassElements.java`)

```
1 /**
2 * This program demonstrates passing individual array
3 * elements as arguments to a method.
4 */
5
6 public class PassElements
7 {
```

```

8 public static void main(String[] args)
9 {
10 // Create an array.
11 int[] numbers = {5, 10, 15, 20, 25, 30, 35, 40};
12
13 // Pass each element to the ShowValue method.
14 for (int index = 0; index < numbers.length; index++)
15 showValue(numbers[index]);
16 }
17
18 /**
19 * The showValue method displays its argument.
20 */
21
22 public static void showValue(int n)
23 {
24 System.out.print(n + " ");
25 }
26 }
```

## Program Output

5 10 15 20 25 30 35 40

The loop in lines 14 and 15 calls the `showValue` method, once for each element in the array. Each time the method is called, an array element is passed to it as an argument. The `showValue` method has an `int` parameter variable named `n` that receives the argument. The method simply displays the contents of `n`. If the method were written to accept the entire array as an argument, however, the parameter would have to be set up differently. In the following method definition, the parameter `array` is declared as an array reference variable. This indicates that the argument will be an array, not a single value.

```

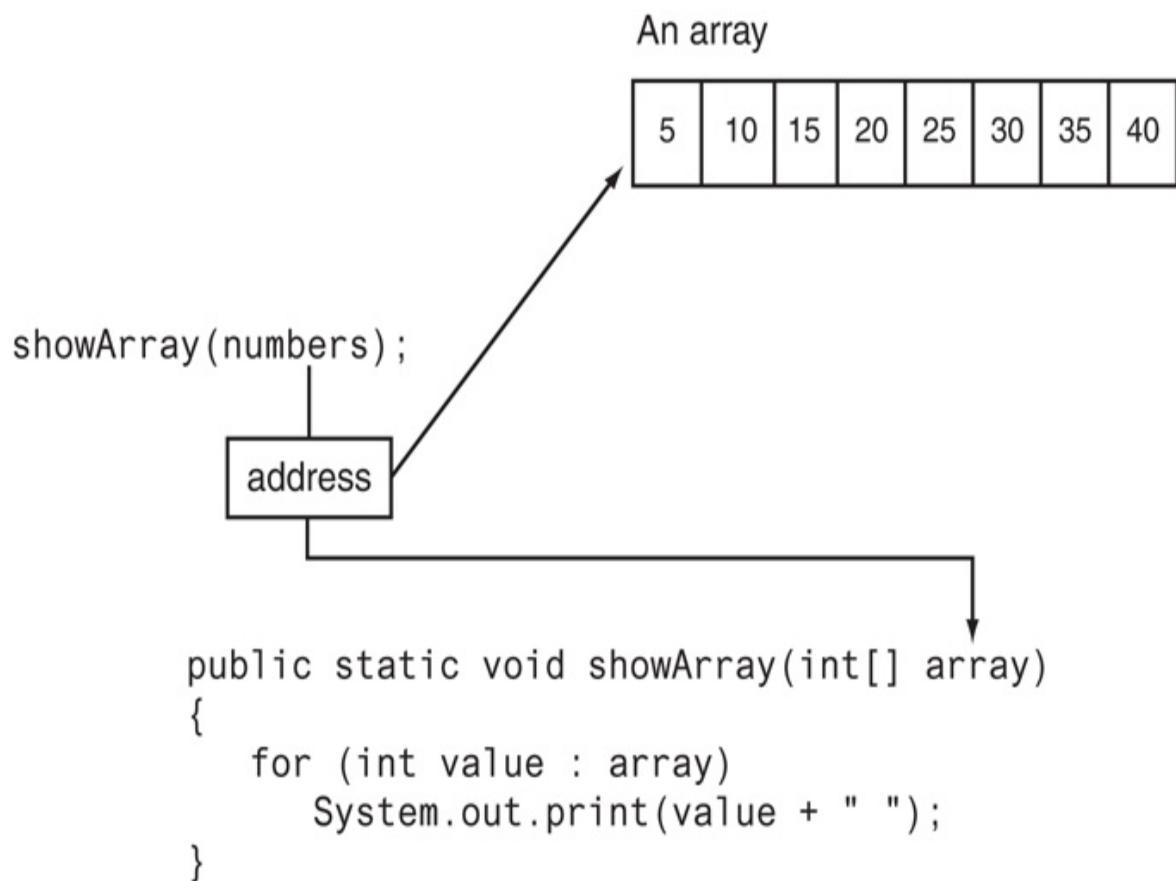
public static void showArray(int[] array)
{
 for (int value : array)
 System.out.print(value + " ");
}
```

When you pass an array as an argument, you simply pass the value in the variable that references the array, as shown here:

```
showArray(numbers);
```

When an entire array is passed into a method, it is passed just as an object is passed: The actual array itself is not passed, but a reference to the array is passed into the parameter. Consequently, this means the method has direct access to the original array. This is illustrated in [Figure 7-12](#).

## Figure 7-12 An array reference passed as an argument



[Figure 7-12 Full Alternative Text](#)

[Code Listing 7-11](#) shows the `showArray` method in use and another method, `getValues`. The `getValues` method accepts an array as an argument. It asks the user to enter a value for each element.

# Code Listing 7-11 (PassArray.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates passing an array as an
5 * argument to a method.
6 */
7
8 public class PassArray
9 {
10 public static void main(String[] args)
11 {
12 final int ARRAY_SIZE = 4; // Size of the array
13
14 // Create an array.
15 int[] numbers = new int[ARRAY_SIZE];
16
17 // Pass the array to the getValues method.
18 getValues(numbers);
19
20 System.out.println("Here are the numbers "
21 "that you entered:");
22
23 // Pass the array to the showArray method.
24 showArray(numbers);
25 }
26
27 /**
28 * The getValues method accepts an array as its
29 * argument. The user is asked to enter a value
30 * for each element.
31 */
32
33 private static void getValues(int[] array)
34 {
35 // Create a Scanner object for keyboard input.
36 Scanner keyboard = new Scanner(System.in);
37
38 System.out.println("Enter a series of " +
39 array.length + " numbers.");
3
40 // Read values into the array.
41 for (int index = 0; index < array.length; index++)
42 {
43
```

```
44 System.out.print("Number " + (index + 1) + ": ");
45 array[index] = keyboard.nextInt();
46 }
47 }
48
49 /**
50 * The showArray method accepts an array as
51 * an argument displays its contents.
52 */
53
54 public static void showArray(int[] array)
55 {
56 // Display the array elements.
57 for (int value : array)
58 System.out.print(value + " ");
59 }
60 }
```

## Program Output with Example Input Shown in Bold

Enter a series of 4 numbers.

Number 1: **2** 

Number 2: **4** 

Number 3: **6** 

Number 4: **8** 

Here are the numbers that you entered:

2 4 6 8



## Checkpoint

1. 7.13 Look at the following method header:

```
public static void myMethod(double[] array)
```

The following code shows an array declaration:

```
final int ARRAY_SIZE = 100;
double[] numbers = new double[ARRAY_SIZE];
```

Write a statement that passes the numbers array to the myMethod method.

2. 7.14 Write a static method named `zero` that accepts an `int` array as an argument and stores the value 0 in each element.

# 7.4 Some Useful Array Algorithms and Operations

## Concept:

In this section, you will see various algorithms written in Java that perform useful operations on arrays.

## Comparing Arrays

In the previous section, you saw that you cannot copy an array by simply assigning its reference variable to another array's reference variable. In addition, you cannot use the `==` operator to compare two array reference variables and determine whether the arrays are equal. For example, the following code appears to compare two arrays, but in reality it does not:

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray) // This is a mistake.
 System.out.println("The arrays are the same.");
else
 System.out.println("The arrays are not the same.");
```

Recall from [Chapter 6](#) that when you use the `==` operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables. Because the two array variables in this code reference different objects in memory, they will contain different addresses. Therefore, the result of the boolean expression `firstArray == secondArray` is `false`, and the code reports that the arrays are not the same.

To compare the contents of two arrays, you must compare the elements of the two arrays. For example, look at the following code.

```

int[] firstArray = { 2, 4, 6, 8, 10 };
int[] secondArray = { 2, 4, 6, 8, 10 };
boolean arraysEqual = true; // Flag variable
int index = 0; // Loop control variable
// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
 arraysEqual = false;
// Next determine whether the elements contain the same data.
while (arraysEqual && index < firstArray.length)
{
 if (firstArray[index] != secondArray[index])
 arraysEqual = false;
 index++;
}

if (arraysEqual)
 System.out.println("The arrays are equal.");
else
 System.out.println("The arrays are not equal.");

```

This code determines whether `firstArray` and `secondArray` contain the same values. A boolean flag variable, `arraysEqual`, which is initialized to `true`, is used to signal whether the arrays are equal. Another variable, `index`, which is initialized to `0`, is used as a loop control variable.

First, this code determines whether the two arrays are the same length. If they are not the same length, then the arrays cannot be equal, so the flag variable `arraysEqual` is set to `false`. Then a `while` loop begins. The loop executes as long as `arraysEqual` is `true` and the control variable `index` is less than `firstArray.length`. During each iteration, it compares a different set of corresponding elements in the arrays. When it finds two corresponding elements that have different values, the flag variable `arraysEqual` is set to `false`. After the loop finishes, an `if` statement examines the `arraysEqual` variable. If the variable is `true`, then the arrays are equal and a message indicating so is displayed. Otherwise, they are not equal, so a different message is displayed.

## Summing the Values in a Numeric Array

To sum the values in an array, you must use a loop with an accumulator variable. The loop adds the value in each array element to the accumulator. For example, assume that the following code appears in a program, and that values have been stored in the `units` array.

```
final int ARRAY_SIZE = 25;
int[] units = new int[ARRAY_SIZE];
```

The following loop adds the values of each element of the `units` array to the `total` variable. When the code is finished, `total` will contain the sum of all of the `units` array's elements.

```
int total = 0; // Initialize accumulator
for (int index = 0; index < units.length; index++)
 total += units[index];
```

You can also use an enhanced for loop to sum the contents of an array, as shown here:

```
int total = 0; // Initialize accumulator
for (int value : units)
 total += value;
```

## Getting the Average of the Values in a Numeric Array

The first step in calculating the average of all the values in an array is to sum the values. The second step is to divide the sum by the number of elements in the array. Assume that the following statement appears in a program, and that values have been stored in the `scores` array:

```
final int ARRAY_SIZE = 10;
double[] scores = new double[ARRAY_SIZE];
```

The following code calculates the average of the values in the `scores` array. When the code completes, the average will be stored in the `average` variable.

```
double total = 0; // Initialize accumulator
```

```

double average; // Will hold the average
// Add up all the values in the array.
for (int index = 0; index < scores.length; index++)
 total += scores[index];
// Calculate the average.
average = total / scores.length;

```

Notice that the last statement, which divides total by scores.length, is not inside the loop. This statement should only execute once, after the loop has finished its iterations.

## Finding the Highest and Lowest Values in a Numeric Array

The algorithms for finding the highest and lowest values in an array are very similar. First, let's look at code for finding the highest value in an array. Assume that the following statement exists in a program, and that values have been stored in the numbers array:

```

final int ARRAY_SIZE = 50;
int[] numbers = new int[ARRAY_SIZE];

```

The code to find the highest value in the array is as follows:

```

int highest = numbers[0];
for (int index = 1; index < numbers.length; index++)
{
 if (numbers[index] > highest)
 highest = numbers[index];
}

```

First, we copy the value in the first array element to the variable highest. Then the loop compares all of the remaining array elements, beginning at subscript 1, to the value in highest. Each time it finds a value in the array greater than highest, it copies that value to highest. When the loop has finished, highest will contain the highest value in the array.

The following code finds the lowest value in the array. As you can see, it is

nearly identical to the code for finding the highest value.

```
int lowest = numbers[0];
for (int index = 1; index < numbers.length; index++)
{
 if (numbers[index] < lowest)
 lowest = numbers[index];
}
```

When the loop has finished, lowest will contain the lowest value in the array.

## The SalesData Class

To demonstrate these algorithms, look at the SalesData class shown in [Code Listing 7-12](#). The class keeps sales amounts for any number of days in an array, which is a private field. Public methods are provided that return the total, average, highest, and lowest amounts of sales. The program in [Code Listing 7-13](#) demonstrates the class.

## Code Listing 7-12 (SalesData.java)

```
1 /**
2 * This class keeps the sales figures for a number of
3 * days in an array and provides methods for getting
4 * the total and average sales, and the highest and
5 * lowest amounts of sales.
6 */
7
8 public class SalesData
9 {
10 private double[] sales; // References the sales data
11
12 /**
13 * The constructor accepts an array as an argument.
14 * The elements in the argument array are copied
15 * to the sales array.
16 */
17
18 public SalesData(double[] s)
19 {
```

```
20 // Create a new array the same length as s.
21 sales = new double[s.length];
22
23 // Copy the values in s to sales.
24 for (int index = 0; index < s.length; index++)
25 sales[index] = s[index];
26 }
27
28 /**
29 * The getTotal method returns the total of the
30 * elements in the sales array.
31 */
32
33 public double getTotal()
34 {
35 double total = 0.0; // Accumulator
36
37 // Add up all the values in the sales array.
38 for (double value : sales)
39 total += value;
40
41 // Return the total.
42 return total;
43 }
44
45 /**
46 * The getAverage method returns the average of the
47 * elements in the sales array.
48 */
49
50 public double getAverage()
51 {
52 return getTotal() / sales.length;
53 }
54
55 /**
56 * The getHighest method returns the highest value
57 * stored in the sales array.
58 */
59
60 public double getHighest()
61 {
62 // Store the first value in the sales array in
63 // the variable highest.
64 double highest = sales[0];
65
66 // Search the array for the highest value.
```

```

67 for (int index = 1; index < sales.length; index++)
68 {
69 if (sales[index] > highest)
70 highest = sales[index];
71 }
72
73 // Return the highest value.
74 return highest;
75 }
76
77 /**
78 * The getLowest method returns the lowest value
79 * stored in the sales array.
80 */
81
82 public double getLowest()
83 {
84 // Store the first value in the sales array in
85 // the variable lowest.
86 double lowest = sales[0];
87
88 // Search the array for the lowest value.
89 for (int index = 1; index < sales.length; index++)
90 {
91 if (sales[index] < lowest)
92 lowest = sales[index];
93 }
94
95 // Return the lowest value.
96 return lowest;
97 }
98 }
```

## Code Listing 7-13 (Sales.java)

```

1 import java.util.Scanner;
2
3 /**
4 * This program gathers sales amounts for the week.
5 * It uses the SalesData class to display the total,
6 * average, highest, and lowest sales amounts.
7 */
8
9 public class Sales
10 {
```

```
11 public static void main(String[] args)
12 {
13 final int ONE_WEEK = 7; // Number of array elements
14
15 // Create an array to hold the sales numbers
16 // for one week.
17 double[] sales = new double[ONE_WEEK];
18
19 // Get the week's sales figures and store them
20 // in the sales array.
21 getValues(sales);
22
23 // Create a SalesData object initialized with the
24 // sales array.
25 SalesData week = new SalesData(sales);
26
27 // Display the total, average, highest, and lowest
28 // sales amounts for the week.
29 System.out.println();
30 System.out.printf("The total sales were $%,.2f\n", week.get
31 System.out.printf("The average sales were $%,.2f\n", week.g
32 System.out.printf("The highest sales were $%,.2f\n", week.g
33 System.out.printf("The lowest sales were $%,.2f\n", week.ge
34 }
35
36 /**
37 * The following method accepts an array as its
38 * argument. The user is asked to enter sales
39 * amounts for each element.
40 */
41
42 private static void getValues(double[] array)
43 {
44 // Create a Scanner object for keyboard input.
45 Scanner keyboard = new Scanner(System.in);
46
47 System.out.println("Enter the sales for each of " +
48 "the following days.");
49
50 // Get the sales for each day in the week.
51 for (int index = 0; index < array.length; index++)
52 {
53 System.out.print("Day " + (index + 1) + ": ");
54 array[index] = keyboard.nextDouble();
55 }
56 }
57 }
```

## Program Output with Example Input Shown in Bold

Enter the sales for each of the following days.

Day 1: **2374.55**

Day 2: **1459.04**

Day 3: **1762.99**

Day 4: **1207.82**

Day 5: **2798.53**

Day 6: **2207.64**

Day 7: **2194.51**

The total sales were \$14,005.08

The average sales were \$2,000.73

The highest sales were \$2,798.53

The lowest sales were \$1,207.82

## In the Spotlight: Creating an Object That Processes an Array



Dr. LaClaire gives a set of exams during the semester in her chemistry class. At the end of the semester, she drops each student's lowest test score before averaging the remaining scores. She has asked you to write a program that will read a student's test scores as input and calculate the average with the lowest score dropped.

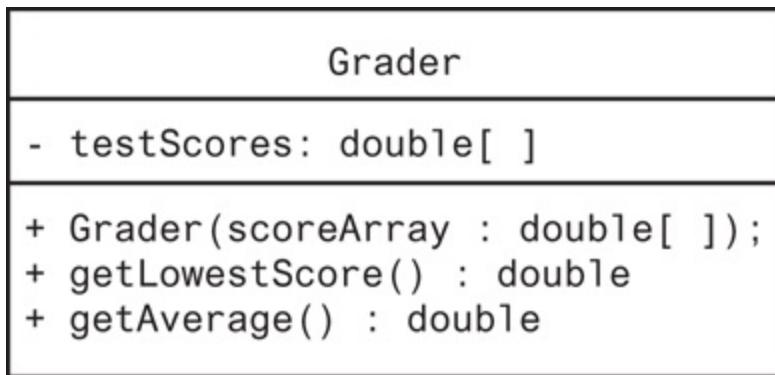
The following pseudocode shows the steps for calculating the average of a set of test scores with the lowest score dropped:

- *Calculate the total of the scores.*
- *Find the lowest score.*

- Subtract the lowest score from the total. This gives the adjusted total.
- Divide the adjusted total by (number of scores – 1). This is the average.

You decide to create a class named `Grader`, with a constructor that accepts a double array of test scores. The `Grader` class will have a method named `getLowestScore` that returns the lowest score in the array, and a method named `getAverage` that returns the average of the test scores with the lowest score dropped. [Figure 7-13](#) shows a UML diagram for the class.

## Figure 7-13 UML diagram for the `Grader` class



[Figure 7-13 Full Alternative Text](#)

[Code Listing 7-14](#) shows the code for the class.

## Code Listing 7-14 (`Grader.java`)

```

1 /**
2 * The Grader class calculates the average
3 * of an array of test scores, with the
4 * lowest score dropped.
5 */
6
7 public class Grader

```

```
8 {
9 // The testScores field is a variable
10 // that will reference an array
11 // of test scores.
12 private double[] testScores;
13
14 /**
15 * The constructor accepts an array of
16 * test scores as an argument.
17 */
18
19 public Grader(double[] scoreArray)
20 {
21 // Assign the array argument to
22 // the testScores field.
23 testScores = scoreArray;
24 }
25
26 /**
27 * The getLowestScore method returns
28 * the lowest test score.
29 */
30
31 public double getLowestScore()
32 {
33 double lowest; // To hold the lowest score
34
35 // Get the first test score in the array.
36 lowest = testScores[0];
37
38 // Step through the rest of the array. When
39 // a value less than lowest is found, assign
40 // it to lowest.
41 for (int index = 1; index < testScores.length; index++)
42 {
43 if (testScores[index] < lowest)
44 lowest = testScores[index];
45 }
46
47 // Return the lowest test score.
48 return lowest;
49 }
50
51 /**
52 * The getAverage method returns the average of the
53 * test scores with the lowest score dropped.
54 */
```

```

55
56 public double getAverage()
57 {
58 double total = 0; // To hold the score total
59 double lowest; // To hold the lowest score
60 double average; // To hold the average
61
62 // If the array contains less than two test
63 // scores, display an error message and set
64 // average to 0.
65 if (testScores.length < 2)
66 {
67 System.out.println("ERROR: You must have at " +
68 "least two test scores!");
69 average = 0;
70 }
71 else
72 {
73 // First, calculate the total of the scores.
74 for (double score : testScores)
75 total += score;
76
77 // Next, get the lowest score.
78 lowest = getLowestScore();
79
80 // Subtract the lowest score from the total.
81 total -= lowest;
82
83 // Get the adjusted average.
84 average = total / (testScores.length - 1);
85 }
86
87 // Return the adjusted average.
88 return average;
89 }
90 }
```

- Line 12 declares a field named `testScores`, which is used to reference a double array of test scores.
- The constructor, in lines 19 through 24, accepts a double array as an argument, which is assigned to the `testScores` field.
- The `getLowestScore` method, in lines 31 through 49, finds the lowest value in the `testScores` array and returns that value.

- The `getAverage` method appears in lines 56 through 89. This method first determines whether there are fewer than two elements in the `testScores` array (in line 65). If that is the case, we cannot drop the lowest score, so an error message is displayed and the `average` variable is set to 0. Otherwise, the code in lines 73 through 84 calculates the average of the test scores with the lowest score dropped and assigns that value to the `average` variable. Line 88 returns the value of the `average` variable.

[Code Listing 7-15](#) shows the program that Dr. LaClaire will use to calculate a student's adjusted average. The program gets a series of test scores, stores those scores in an array, and uses an instance of the `Grader` class to calculate the average.

## Code Listing 7-15 (`CalcAverage.java`)

```

1 import java.util.Scanner;
2
3 /**
4 * This program gets a set of test scores and
5 * uses the Grader class to calculate the average
6 * with the lowest score dropped.
7 */
8
9 public class CalcAverage
10 {
11 public static void main(String[] args)
12 {
13 int numScores; // To hold the number of scores
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the number of test scores.
19 System.out.print("How many test scores do you have? ");
20 numScores = keyboard.nextInt();
21
22 // Create an array to hold the test scores.
23 double[] scores = new double[numScores];

```

```

24
25 // Get the test scores and store them
26 // in the scores array.
27 for (int index = 0; index < numScores; index++)
28 {
29 System.out.print("Enter score #" +
30 (index + 1) + ": ");
31 scores[index] = keyboard.nextDouble();
32 }
33
34 // Create a Grader object, passing the
35 // scores array as an argument to the
36 // constructor.
37 Grader myGrader = new Grader(scores);
38
39 // Display the adjusted average.
40 System.out.println("Your adjusted average is " +
41 myGrader.getAverage());
42
43 // Display the lowest score.
44 System.out.println("Your lowest test score was " +
45 myGrader.getLowestScore());
46 }
47 }
```

## Program Output with Example Input Shown in Bold

```

How many test scores do you have? 4 
Enter scores #1: 100 
Enter scores #2: 100 
Enter scores #3: 40 
Enter scores #4: 100 
Your adjusted average is 100.0
Your lowest test score was 40.0
```

# Partially Filled Arrays

Sometimes you need to store a series of items in an array, but you don't know the number of items that there are. As a result, you don't know the exact number of elements needed for the array. One solution is to make the array large enough to hold the largest possible number of items. This can lead to

another problem, however: If the actual number of items stored in the array is less than the number of elements, the array will be only partially filled. When you process a partially filled array, you must only process the elements that contain valid data items.

A partially filled array is normally used with an accompanying integer variable that holds the number of items stored in the array. For example, suppose a program uses the following code to create an array with 100 elements, and an `int` variable named `count` that will hold the number of items stored in the array:

```
final int ARRAY_SIZE = 100;
int[] array = new int[ARRAY_SIZE];
int count = 0;
```

Each time we add an item to the array, we must increment `count`. The following code demonstrates this:

```
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter a number or -1 to quit: ");
number = keyboard.nextInt();
while (number != -1 && count < array.length)
{
 array[count] = number;
 count++;
 System.out.print("Enter a number or -1 to quit: ");
 number = keyboard.nextInt();
}
```

Each iteration of this sentinel-controlled loop allows the user to enter a number to be stored in the array, or -1 to quit. The `count` variable is used as the subscript of the next available element in the array, then incremented. When the user enters -1, or `count` reaches the size of the array, the loop stops. The following code displays all of the valid items in the array:

```
for (int index = 0; index < count; index++)
{
 System.out.println(array[index]);
}
```

Notice this code uses `count` to determine the maximum array subscript to use.



## Note:

If a partially filled array is passed as an argument to a method, the variable that holds the count of items in the array must also be passed as an argument. Otherwise, the method will not be able to determine the number of items stored in the array.

# Working with Arrays and Files

Saving the contents of an array to a file is a straightforward procedure: Use a loop to step through each element of the array, writing its contents to the file. For example, assume a program defines an array as follows:

```
int[] numbers = { 10, 20, 30, 40, 50 };
```

The following code opens a file named *Values.txt* and writes the contents of each element of the *numbers* array to the file:

```
int[] numbers = { 10, 20, 30, 40, 50 };
// Open the file.
PrintWriter outputFile = new PrintWriter("Values.txt");
// Write the array elements to the file.
for (int index = 0; index < numbers.length; index++)
 outputFile.println(numbers[index]);
// Close the file.
outputFile.close();
```

The following code demonstrates how to open the *Values.txt* file and read its contents back into the *numbers* array:

```
final int SIZE = 5;
int[] numbers = new int[SIZE];
int index = 0; // Loop control variable

// Open the file.
File file = new File("Values.txt");
Scanner inputFile = new Scanner(file);
```

```
// Read the file contents into the array.
while (inputFile.hasNext() && index < numbers.length)
{
 numbers[index] = inputFile.nextInt();
 index++;
}

// Close the file.
inputFile.close();
```

The file is opened, then a while loop reads all of the values from the file into the numbers array. The loop repeats as long as `inputFile.hasNext()` returns true, and `index` is less than `numbers.length`. The `inputFile.hasNext()` method is called to make sure there is a value remaining in the file. This prevents an error in case the file does not contain enough values to fill the array. The second condition (`index < numbers.length`) prevents the loop from writing outside the array boundaries.

# 7.5 Returning Arrays from Methods

## Concept:

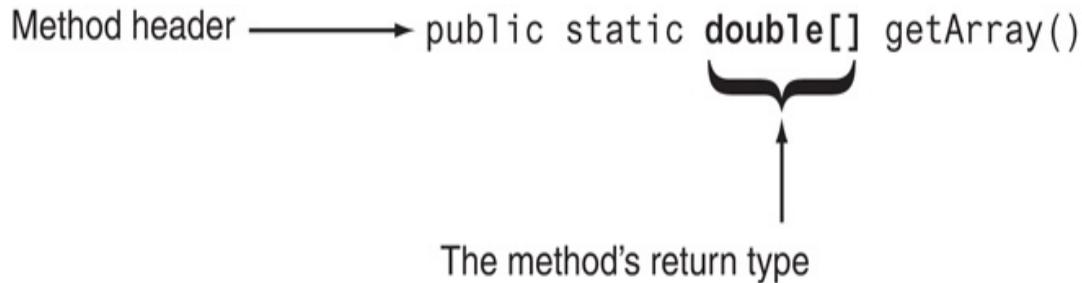
In addition to accepting arrays as arguments, methods may also return arrays.

A method can return a reference to an array. To do so, the return type of the method must be properly declared. For example, look at the following method definition:

```
public static double[] getArray()
{
 double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
 return array;
}
```

The `getArray` method is a public static method that returns an array of doubles. Notice that the return type listed in the method header is `double[]`. The method header is illustrated in [Figure 7-14](#). It indicates that the method returns a reference to a double array.

## Figure 7-14 Array reference return type



Inside the method an array of doubles is created, initialized with some values, and referenced by the array variable. The return statement then returns the array variable. By returning the array variable, the method is returning a reference to the array. The method's return value can be stored in any compatible reference variable, as demonstrated in [Code Listing 7-16](#).

## Code Listing 7-16 (`ReturnArray.java`)

```
1 /**
2 * This program demonstrates how a reference to an
3 * array can be returned from a method.
4 */
5
6 public class ReturnArray
7 {
8 public static void main(String[] args)
9 {
10 double[] values;
11
12 // Let values reference the array returned
13 // from the getArray method.
14 values = getArray();
15
16 // Display the values in the array.
17 for (double num : values)
18 System.out.println(num);
19 }
20
21 /**
22 * The getArray method returns a reference to
23 * an array of doubles.
24 */
25
26 public static double[] getArray()
27 {
28 double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
29 return array;
30 }
31 }
```

### Program Output

1.2 2.3 4.5 6.7 8.9

The statement in line 14 assigns the array created in `getArray` to the array variable `values`. The `for` loop in lines 17 and 18 then displays the value of each element of the `values` array.

We could easily modify the `getArray` method to create an array of a specified size and fill it with values entered by the user. The following shows an example:

```
public static double[] getArray(int size)
{
 // Create an array of the specified size.
 double[] array = new double[size];

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

 System.out.println("Enter a series of " +
 array.length + " numbers.");

 // Get values from the user for the array.
 for (int index = 0; index < array.length; index++)
 {
 System.out.print("Number " + (index + 1) + ": ");
 array[index] = keyboard.nextInt();
 }

 // Return the array.
 return array;
}
```

## 7.6 String Arrays

### Concept:

An array of `String` objects may be created, but if the array is uninitialized, each `String` in the array must be created individually.

Java also allows you to create arrays of `String` objects. Here is a statement that creates an array of `String` objects initialized with values:

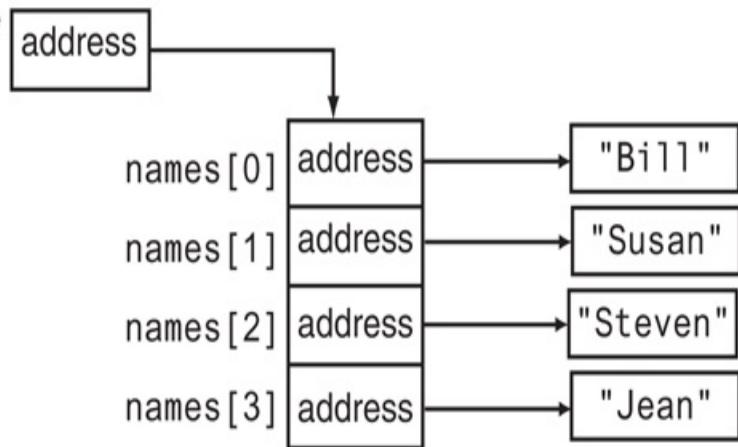
```
String[] names = { "Bill", "Susan", "Steven", "Jean" };
```

In memory, an array of `String` objects is arranged differently than an array of a primitive data type. To use a `String` object, you must have a reference to the `String` object. So, an array of `String` objects is really an array of references to `String` objects. [Figure 7-15](#) illustrates how the `names` variable will reference an array of references to `String` objects.

**Figure 7-15 The `names` variable references a `String` array**

A String array is  
an array of references  
to String objects.

The names variable holds the  
address of a String array.



[Figure 7-15 Full Alternative Text](#)

Each element in the names array is a reference to a String object. The names[0] element references a String object containing “Bill”, the names[1] element references a String object containing “Susan”, and so forth. The program in [Code Listing 7-17](#) demonstrates an array of String objects.

## Code Listing 7-17 (MonthDays.java)

```
1 /**
2 * This program demonstrates an array of String objects.
3 */
4
5 public class MonthDays
6 {
7 public static void main(String[] args)
8 {
9 // Create an array of Strings containing the names
10 // of the months.
11 String[] months = { "January", "February", "March",
12 "April", "May", "June", "July",
13 "August", "September", "October",
14 "November", "December" };
15
16 // Create an array of ints containing the numbers
17 // of days in each month.
```

```
18 int[] days = { 31, 28, 31, 30, 31, 30, 31,
19 31, 30, 31, 30, 31 };
20
21 // Display the months and the days in each.
22 for (int index = 0; index < months.length; index++)
23 {
24 System.out.println(months[index] + " has " +
25 days[index] + " days.");
26 }
27 }
28 }
```

## Program Output

```
January has 31 days.
February has 28 days.
March has 31 days.
April has 30 days.
May has 31 days.
June has 30 days.
July has 31 days.
August has 31 days.
September has 30 days.
October has 31 days.
November has 30 days.
December has 31 days.
```

As with the primitive data types, an initialization list automatically causes an array of `String` objects to be created in memory. If you do not provide an initialization list, you must use the `new` key word to create the array. Here is an example:

```
final int ARRAY_SIZE = 4;
String[] names = new String[ARRAY_SIZE];
```

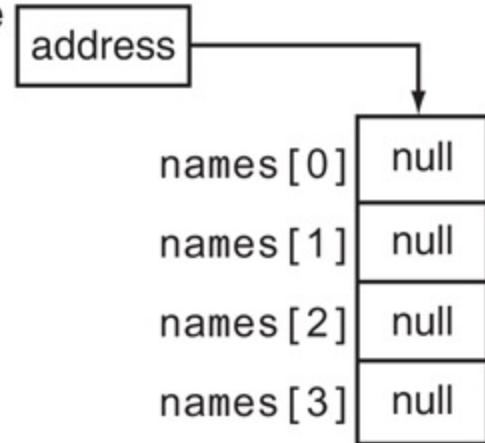
This statement creates an array of four references to `String` objects, as shown in [Figure 7-16](#). Notice that the array is an array of four uninitialized `String` references. Because they do not reference any objects, they are set to `null`.

## Figure 7-16 An uninitialized

# String array

A String array is an array of references to String objects.

The names variable holds the address of a String array.



[Figure 7-16 Full Alternative Text](#)

When you create an uninitialized array of `String` objects, you must assign a value to each element in the array. Here is an example:

```
final int ARRAY_SIZE = 4;
String[] names = new String[ARRAY_SIZE];
names[0] = "Bill";
names[1] = "Susan";
names[2] = "Steven";
names[3] = "Jean";
```

After these statements execute, each element of the `names` array will reference a `String` object.

## Calling `String` Methods from an Array Element

Recall from [Chapter 2](#) that `String` objects have several methods. For

example, the `toUpperCase` method returns the uppercase equivalent of a `String` object. Because each element of a `String` array is a `String` object, you can use an element to call a `String` method. For example, the following statement calls the `toUpperCase` method from element 0 of the `names` array:

```
System.out.println(names[0].toUpperCase());
```

The program in [Code Listing 7-18](#) uses a loop to call the `toUpperCase` method from each element of the `names` array.

## Code Listing 7-18 (`StringArrayMethods.java`)

```
1 /**
2 * This program demonstrates the toUpperCase method
3 * being called from the elements of a String array.
4 */
5
6 public class StringArrayMethods
7 {
8 public static void main(String[] args)
9 {
10 // Create an array of Strings.
11 String[] names = { "Bill", "Susan",
12 "Steven", "Jean" };
13
14 // Display each string in the names array
15 // in uppercase.
16 for (int index = 0; index < names.length; index++)
17 System.out.println(names[index].toUpperCase());
18 }
19 }
```

### Program Output

```
BILL
SUSAN
STEVEN
JEAN
```

## Tip:

Arrays have a field named `length`, and `String` objects have a method named `length`. When working with `String` arrays, do not confuse the two. The following loop displays the length of each string held in a `String` array. Note that the loop uses both the array's `length` field and each element's `length` method.

```
for (int index = 0; index < names.length; index++)
 System.out.println(names[index].length());
```

Because the array's `length` member is a field, you do not write a set of parentheses after its name. You do write the parentheses after the name of the `String` class's `length` method.



## Checkpoint

### 1. 7.15

1. Write a statement that declares a `String` array initialized with the following strings: “Mercury”, “Venus”, “Earth”, and “Mars”.
2. Write a loop that displays the contents of each element in the array you declared in [Checkpoint 7.15a](#).
3. Write a loop that displays the first character of the strings stored in each element of the array you declared in [Checkpoint 7.15a](#). (Hint: Use the `String` class's `charAt` method discussed in [Chapter 2](#).)

# 7.7 Arrays of Objects

## Concept:

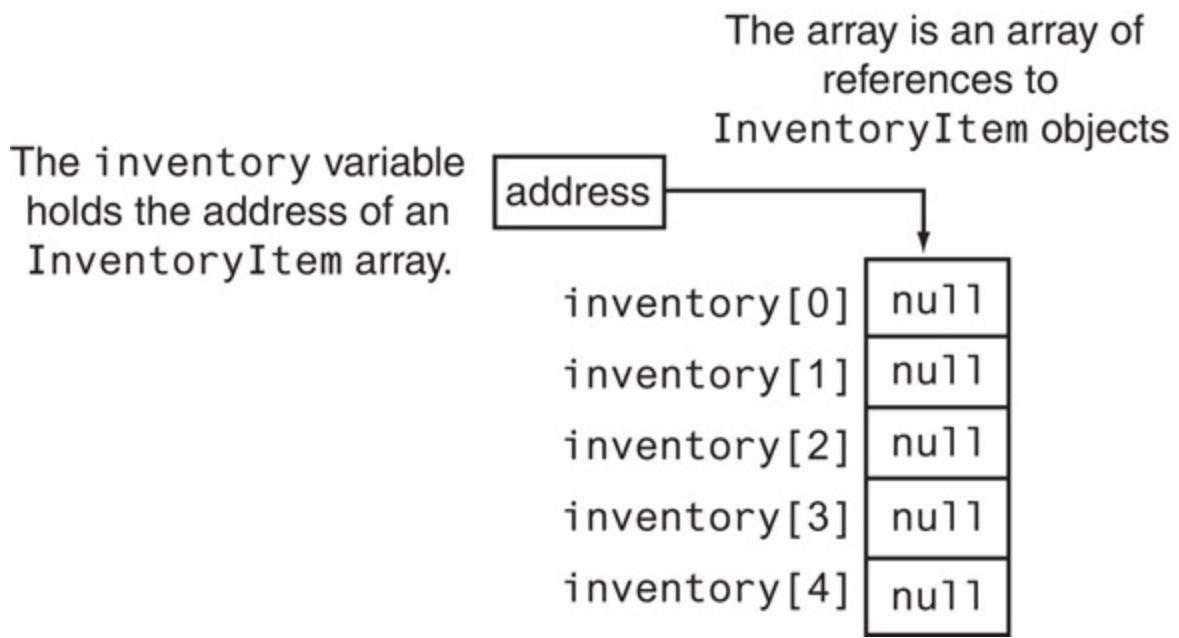
You may create arrays of objects that are instances of classes that you have written.

Like any other data type, you can create arrays of class objects. For example, recall the `InventoryItem` class introduced in [Chapter 6](#). An array of `InventoryItem` objects could be created to represent a business's inventory records. Here is a statement that declares an array of five `InventoryItem` objects:

```
final int NUM_ITEMS = 5;
InventoryItem[] inventory = new InventoryItem[NUM_ITEMS];
```

The variable that references the array is named `inventory`. As with `String` arrays, each element in this array is a reference variable, as illustrated in [Figure 7-17](#).

**Figure 7-17 The `inventory` variable references an array of references**



[Figure 7-17 Full Alternative Text](#)

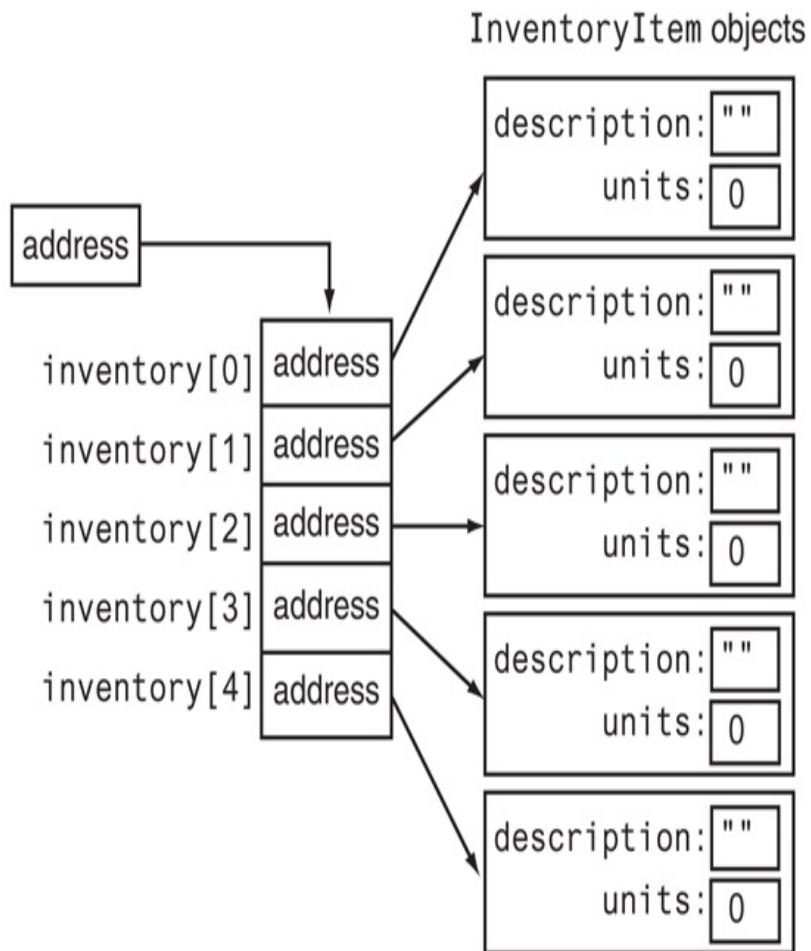
Notice from the figure that each element of the array is initialized with the `null` value. This indicates that the array elements do not yet reference objects. You must create the objects that each element will reference. The following code uses a loop to create objects for each element:

```
for (int index = 0; index < inventory.length; index++)
 inventory[index] = new InventoryItem();
```

In this code, the no-arg constructor is called for each object. Recall that the `InventoryItem` class has a no-arg constructor that assigns an empty string ("") to the `description` field and 0 to the `units` field. After the loop executes, each element of the `inventory` array will reference an object, as shown in [Figure 7-18](#).

## Figure 7-18 Each element of the array references an object

The inventory variable holds the address of an InventoryItem array.



[Figure 7-18 Full Alternative Text](#)

Objects in an array are accessed with subscripts, just like any other data type in an array. For example, the following code calls the `setDescription` and `setUnits` methods of the element `inventory[2]`:

```
inventory[2].setDescription("Wrench");
inventory[2].setUnits(20);
```

[Code Listing 7-19](#) shows a complete program that uses an array of objects.

## Code Listing 7-19 (ObjectArray.java)

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

```
2 /**
3 * This program works with an array of InventoryItem objects.
4 */
5
6
7 public class ObjectArray
8 {
9 public static void main(String[] args)
10 {
11 final int NUM_ITEMS = 3; // Number of items
12
13 // Create an InventoryItem array.
14 InventoryItem[] inventory = new InventoryItem[NUM_ITEMS];
15
16 // Call the getItems method to get data for each element.
17 getItems(inventory);
18
19 System.out.println("You entered the following:");
20
21 // Display the data that the user entered.
22 for (int index = 0; index < inventory.length; index++)
23 {
24 System.out.println("Item " + (index + 1));
25 System.out.println("Description: " +
26 inventory[index].getDescription());
27 System.out.println("Units: " +
28 inventory[index].getUnits());
29 System.out.println();
30 }
31 }
32
33 /**
34 * The getItems method accepts an InventoryItem array as
35 * an argument. The user enters data for each element.
36 */
37
38 private static void getItems(InventoryItem[] array)
39 {
40 String description; // Item description
41 int units; // Number of units on hand
42
43 // Create a Scanner object for keyboard input.
44 Scanner keyboard = new Scanner(System.in);
45
46 System.out.println("Enter data for " + array.length +
47 " inventory items.");
48
49 // Get data for the array.
```

```
50 for (int index = 0; index < array.length; index++)
51 {
52 // Get an item's description.
53 System.out.print("Enter the description for " +
54 "item " + (index + 1) + ": ");
55 description = keyboard.nextLine();
56
57 // Get the number of units.
58 System.out.print("Enter the units for " +
59 "item " + (index + 1) + ": ");
60 units = keyboard.nextInt();
61
62 // Consume the remaining newline.
63 keyboard.nextLine();
64
65 // Create an InventoryItem object initialized with
66 // the data and store the object in the array.
67 array[index] = new InventoryItem(description, units);
68
69 // Display a blank line before going on.
70 System.out.println();
71 }
72 }
73 }
```

## Program Output with Example Input Shown in Bold

Enter data for 3 inventory items.

Enter the description for item 1: **Wrench** 

Enter the units for item 1: **20** 

Enter the description for item 2: **Hammer** 

Enter the units for item 2: **15** 

Enter the description for item 3: **Pliers** 

Enter the units for item 3: **18** 

You entered the following:

Item 1

Description: Wrench

Units: 20

Item 2

Description: Hammer

Units: 15

Item 3

Description: Pliers

Units: 18



## Checkpoint

1. 7.16 Recall that we discussed a `Rectangle` class in [Chapter 3](#). Write code that declares a `Rectangle` array with five elements. Instantiate each element with a `Rectangle` object. Use the `Rectangle` constructor to initialize each object with values for the `length` and `width` fields.

# 7.8 The Sequential Search Algorithm

## Concept:

A search algorithm is a method of locating a specific item in a larger collection of data. This section discusses the sequential search algorithm, which is a simple technique for searching the contents of an array.

It is very common for programs not only to store and process information stored in arrays, but to search arrays for specific items. This section shows you how to use the simplest of all search algorithms, the sequential search.

The *sequential search algorithm* uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm unsuccessfully searches to the end of the array.

The `SearchArray` class shown in [Code Listing 7-20](#) uses a static method, `sequentialSearch`, to find a value in an integer array. The argument `array` is searched for an occurrence of the number stored in `value`. If the number is found, its array subscript is returned. Otherwise, `-1` is returned, indicating the value did not appear in the array.

## Code Listing 7-20 (`SearchArray.java`)

```
1 /**
```

```
2 * This class's sequentialSearch method searches an
3 * int array for a specified value.
4 */
5
6 public class SearchArray
7 {
8 /**
9 * The sequentialSearch method searches array for
10 * value. If value is found in array, the element's
11 * subscript is returned. Otherwise, -1 is returned.
12 */
13
14 public static int sequentialSearch(int[] array, int value)
15 {
16 int index, // Loop control variable
17 element; // Element the value is found at
18 boolean found; // Flag indicating search results
19
20 // Element 0 is the starting point of the search.
21 index = 0;
22
23 // Store the default values for element and found.
24 element = -1;
25 found = false;
26
27 // Search the array.
28 while (!found && index < array.length)
29 {
30 // Does this element have the value?
31 if (array[index] == value)
32 {
33 found = true; // Indicate the value is found.
34 element = index; // Save the subscript of the value.
35 }
36
37 // Increment index so we can look at the next element.
38 index++;
39 }
40
41 // Return either the subscript of the value (if found)
42 // or -1 to indicate the value was not found.
43 return element;
44 }
45 }
```



## Note:

The reason -1 is returned when the search value is not found in the array is because -1 is not a valid subscript.

[Code Listing 7-21](#) is a complete program that uses the `SearchArray` class. It searches the five-element array `tests` to find a score of 100.

# Code Listing 7-21 (`TestSearch.java`)

```
1 /**
2 * This program demonstrates the SearchArray class's
3 * sequentialSearch method.
4 */
5
6 public class TestSearch
7 {
8 public static void main(String[] args)
9 {
10 int results; // Results of the search
11
12 // Create an array of values.
13 int[] tests = { 87, 75, 98, 100, 82 };
14
15 // Search the array for the value 100.
16 results = SearchArray.sequentialSearch(tests, 100);
17
18 // Determine whether 100 was found in the array.
19 if (results == -1)
20 {
21 // -1 indicates the value was not found.
22 System.out.println("You did not earn 100 " +
23 "on any test.");
24 }
25 else
26 {
27 // results holds the subscript of the value 100.
28 System.out.println("You earned 100 on " +
29 "test " + (results + 1));
30 }
31 }
32 }
```

```
30 }
31 }
32 }
```

## Program Output

You earned 100 on test 4

See the PinTester Class Case Study for another example using arrays. Also, see the Case Study on Parallel Arrays to learn about another programming technique using arrays. These case studies are available on the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

# 7.9 The Selection Sort and the Binary Search Algorithms

## Concept:

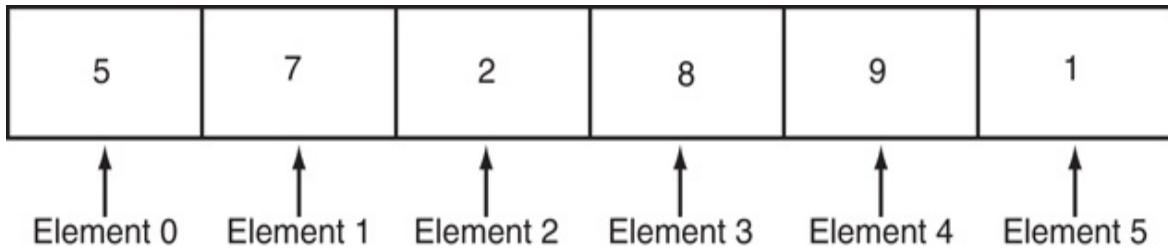
A sorting algorithm is used to arrange data into some order. A search algorithm is a method of locating a specific item in a larger collection of data. The selection sort and the binary search are popular sorting and searching algorithms.

## The Selection Sort Algorithm

Often the data in an array must be sorted in some order. Customer lists, for instance, are commonly sorted in alphabetical order. Student grades might be sorted from highest to lowest. Product codes could be sorted so all the products of the same color are stored together. In this section, we explore how to write a sorting algorithm. A *sorting algorithm* is a technique for scanning through an array and rearranging its contents in some specific order. The algorithm we will explore is called the *selection sort*.

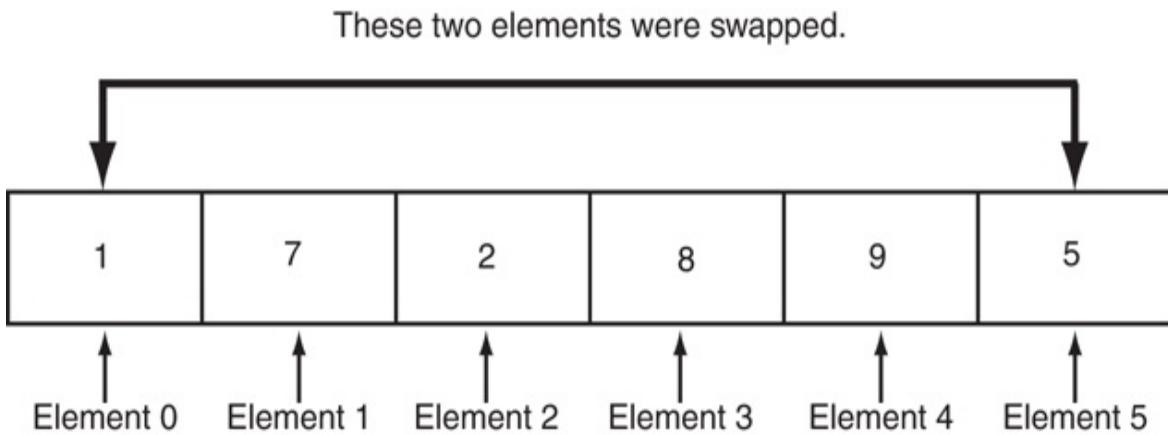
The *selection sort* works like this: The smallest value in the array is located and moved to element 0. Then the next smallest value is located and moved to element 1. This process continues until all of the elements have been placed in their proper order. Let's see how the selection sort works when arranging the elements of the following array in [Figure 7-19](#).

## Figure 7-19 Values in an array



The selection sort scans the array, starting at element 0, and locates the element with the smallest value. The contents of this element are then swapped with the contents of element 0. In this example, the 1 stored in element 5 is swapped with the 5 stored in element 0. After the exchange, the array would appear as shown in [Figure 7-20](#).

## Figure 7-20 Values in array after first swap

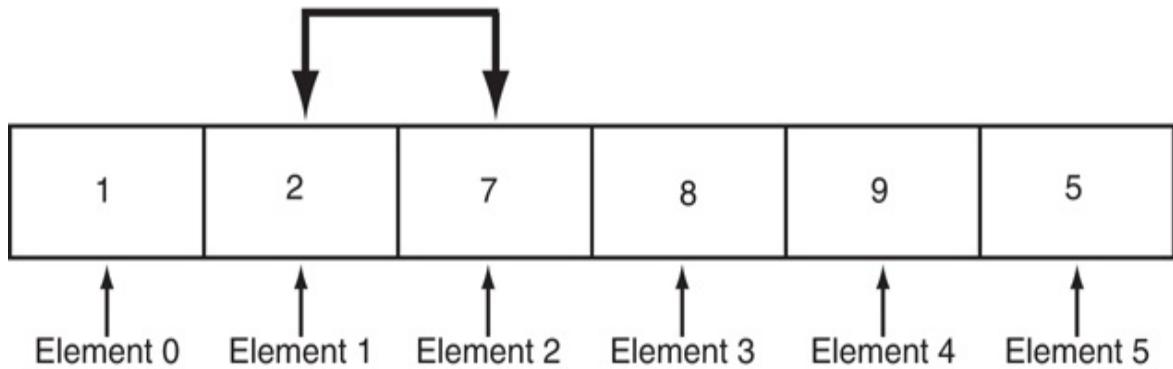


[Figure 7-20 Full Alternative Text](#)

The algorithm then repeats the process, but because element 0 already contains the smallest value in the array, it can be left out of the procedure. This time, the algorithm begins the scan at element 1. In this example, the contents of element 2 are exchanged with that of element 1. The array would then appear as shown in [Figure 7-21](#).

# Figure 7-21 Values in array after second swap

These two elements were swapped.

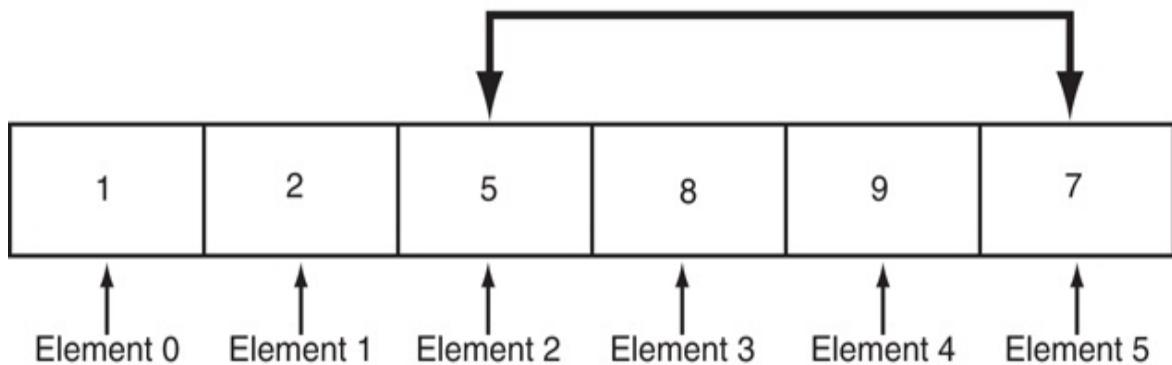


[Figure 7-21 Full Alternative Text](#)

Once again the process is repeated, but this time the scan begins at element 2. The algorithm will find that element 5 contains the next smallest value. This element's value is swapped with that of element 2, causing the array to appear as shown in [Figure 7-22](#).

# Figure 7-22 Values in array after third swap

These two elements were swapped.

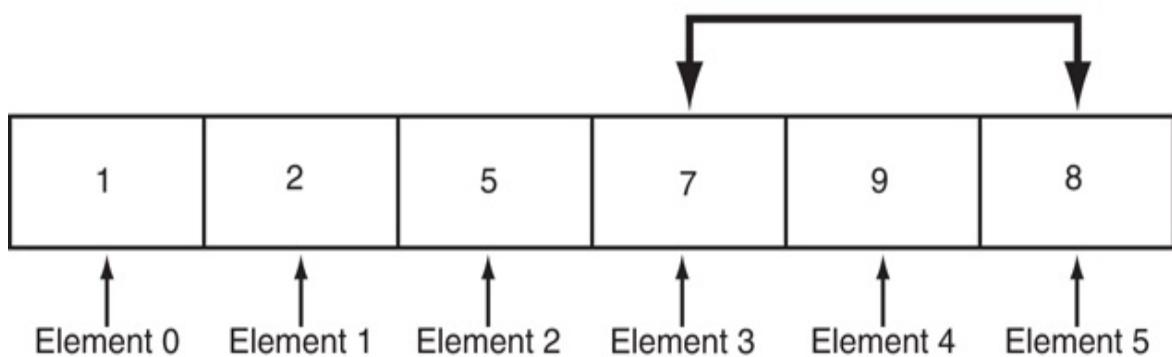


[Figure 7-22 Full Alternative Text](#)

Next, the scanning begins at element 3. Its value is swapped with that of element 5, causing the array to appear as shown in [Figure 7-23](#).

## Figure 7-23 Values in array after fourth swap

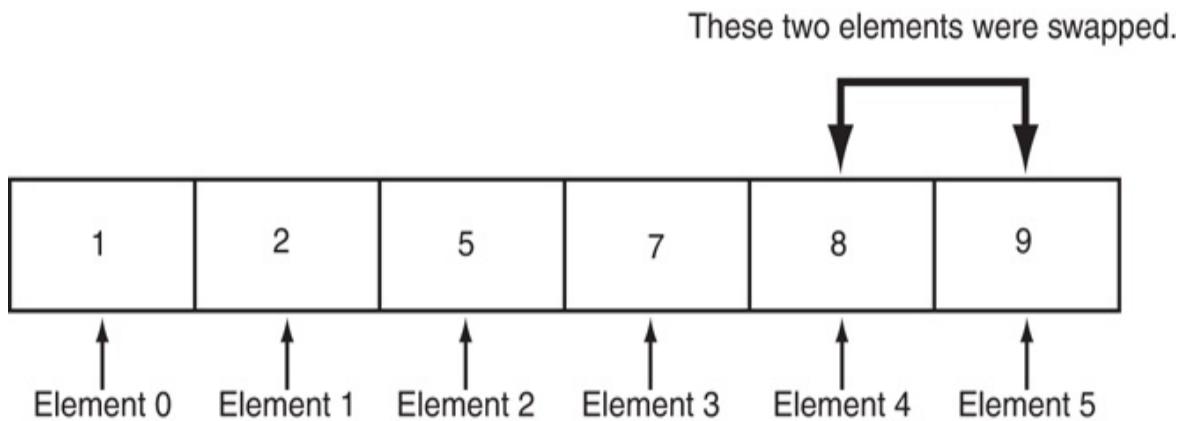
These two elements were swapped.



[Figure 7-23 Full Alternative Text](#)

At this point, there are only two elements left to sort. The algorithm finds that the value in element 5 is smaller than that of element 4, so the two are swapped. This puts the array in its final arrangement as shown in [Figure 7-24](#).

# Figure 7-24 Values in array after fifth swap



[Figure 7-24 Full Alternative Text](#)

Here is the selection sort algorithm in pseudocode:

```
For startScan is each subscript in array from 0 through the next-
 Set minValue variable to startScan.
 Set minValue variable to array[startScan].
 For index is each subscript in array from (startScan + 1) thr
 subscript
 If array[index] is less than minValue
 Set minValue to array[index].
 Set minIndex to index.
 End If.
 Increment index.
 End For.
 Set array[minIndex] to array[startScan].
 Set array[startScan] to minValue.
End For.
```

The following static method performs a selection sort on an integer array. The array that is passed as an argument is sorted in ascending order.

```
public static void selectionSort(int[] array)
{
 int startScan, index, minIndex, minValue;
 for (startScan = 0; startScan < (array.length-1); startScan++)
```

```

{
 minIndex = startScan;
 minValue = array[startScan];
 for(index = startScan + 1; index < array.length; index++)
 {
 if (array[index] < minValue)
 {
 minValue = array[index];
 minIndex = index;
 }
 }
 array[minIndex] = array[startScan];
 array[startScan] = minValue;
}
}

```

The selectionSort method is in the ArrayTools class, which is available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). The program in [Code Listing 7-22](#) demonstrates it.

## Code Listing 7-22 (SelectionSortDemo.java)

```

1 /**
2 * This program demonstrates the selectionSort method
3 * in the ArrayTools class.
4 */
5
6 public class SelectionSortDemo
7 {
8 public static void main(String[] args)
9 {
10 // Create an array of unsorted values.
11 int[] values = {5, 7, 2, 8, 9, 1};
12
13 // Display the unsorted array.
14 System.out.println("The unsorted values are:");
15 for (int index = 0; index < values.length; index++)
16 System.out.print(values[index] + " ");
17 System.out.println();
18
19 // Sort the array.

```

```
20 ArrayTools.selectionSort(values);
21
22 // Display the sorted array.
23 System.out.println("The sorted values are:");
24 for (int index = 0; index < values.length; index++)
25 System.out.print(values[index] + " ");
26 }
27 }
```

## Program Output

```
The unsorted values are:
5 7 2 8 9 1
The sorted values are:
1 2 5 7 8 9
```

# The Binary Search Algorithm

This chapter previously presented the sequential search algorithm for searching an array. The advantage of the sequential search is its simplicity. It is easy to understand and implement. Furthermore, it doesn't require the data in the array to be stored in any particular order. Its disadvantage, however, is its inefficiency. If the array being searched contains 20,000 elements, the algorithm will have to look at all 20,000 elements to find a value stored in the last element. In an average case, an item is just as likely to be found near the end of the array as near the beginning. Typically, for an array of  $N$  items, the sequential search will locate an item in  $N/2$  attempts. If an array has 50,000 elements, the sequential search will make a comparison with 25,000 of them in a typical case.

This is assuming, of course, that the search item is consistently found in the array. ( $N/2$  is the average number of comparisons. The maximum number of comparisons is always  $N$ .) When the sequential search fails to locate an item, it must make a comparison with every element in the array. As the number of failed search attempts increases, so does the average number of comparisons. Obviously, the sequential search should not be used on large arrays if speed is important.

The *binary search* is a clever algorithm that is much more efficient than the sequential search. Its only requirement is that the values in the array must be sorted in ascending order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over. Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater, then the desired value (if it is in the list) will be found somewhere in the first half of the array. If it is less, then the desired value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array's elements have been eliminated from further searching.

If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value. For instance, if the last half of the array is to be searched, the algorithm tests its middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found, or there are no more elements to test. Here is the pseudocode for a method that performs a binary search on an array:

```
Set first to 0.
Set last to the last subscript in the array.
Set position to -1.
Set found to false.
While found is not true and first is less than or equal to last
 Set middle to the subscript halfway between array[first] and
 If array[middle] equals the desired value
 Set found to true.
 Set position to middle.
 Else If array[middle] is greater than the desired value
 Set last to middle -1.
 Else
 Set first to middle +1.
 End If.
End While.
Return position.
```

This algorithm uses three variables to mark positions within the array: `first`, `last`, and `middle`. The `first` and `last` variables mark the boundaries of the portion of the array currently being searched. They are initialized with the

subscripts of the array's first and last elements. The subscript of the element halfway between first and last is calculated and stored in the middle variable. If the element in the middle of the array does not contain the search value, the first or last variables are adjusted so only the top or bottom half of the array is searched during the next iteration. This cuts the portion of the array being searched in half each time the loop fails to locate the search value.

The following static method performs a binary search on an integer array. The first parameter, array, is searched for an occurrence of the number stored in value. If the number is found, its array subscript is returned. Otherwise, -1 is returned, indicating the value did not appear in the array.

```
public static int binarySearch(int[] array, int value)
{
 int first, // First array element
 last, // Last array element
 middle, // Midpoint of search
 position; // Position of search value
 boolean found; // Flag

 // Set the initial values.
 first = 0;
 last = array.length - 1;
 position = -1;
 found = false;

 // Search for the value.
 while (!found && first <= last)
 {
 middle = (first + last) / 2; // Calculate midpoint
 if (array[middle] == value) // If value is found at mid
 {
 found = true;
 position = middle;
 }
 else if (array[middle] > value) // If value is in lower half
 last = middle - 1;
 else
 first = middle + 1; // If value is in upper half
 }

 // Return the position of the item, or -1
 // if it was not found.
```

```
 return position;
}
```

The `binarySearch` method is in the `ArrayTools` class, which is available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). The program in [Code Listing 7-23](#) demonstrates it. Note the values in the array are already sorted in ascending order.

## Code Listing 7-23 (BinarySearchDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the binary search method in
5 * the ArrayTools class.
6 */
7
8 public class BinarySearchDemo
9 {
10 public static void main(String[] args)
11 {
12 int result, // Result of the search
13 searchValue; // Value to search for
14 String again; // Indicates whether to search again
15
16 // Create a Scanner object for keyboard input.
17 Scanner keyboard = new Scanner(System.in);
18
19 // The values in the following array are sorted
20 // in ascending order.
21 int numbers[] = {101, 142, 147, 189, 199, 207, 222,
22 234, 289, 296, 310, 319, 388, 394,
23 417, 429, 447, 521, 536, 600};
24
25 do
26 {
27 // Get a value to search for.
28 System.out.print("Enter a value to search for: ");
29 searchValue = keyboard.nextInt();
30 }
```

```
31 // Search for the value
32 result = ArrayTools.binarySearch(numbers, searchValue);
33
34 // Display the results.
35 if (result == -1)
36 System.out.println(searchValue + " was not found.");
37 else
38 {
39 System.out.println(searchValue + " was found at " +
40 "element " + result);
41 }
42
43 // Consume the remaining newline.
44 keyboard.nextLine();
45
46 // Does the user want to search again?
47 System.out.print("Do you want to search again? (Y or N): ");
48 again = keyboard.nextLine();
49
50 } while (again.charAt(0) == 'y' || again.charAt(0) == 'Y');
51 }
52 }
```

## Program Output with Example Input Shown in Bold

```
Enter a value to search for: 296
296 was found at element 9

Do you want to search again? (Y or N): y
Enter a value to search for: 600
600 was found at element 19

Do you want to search again? (Y or N): y
Enter a value to search for: 101
101 was found at element 0

Do you want to search again? (Y or N): y
Enter a value to search for: 207
207 was found at element 5

Do you want to search again? (Y or N): y
Enter a value to search for: 999
999 was not found.

Do you want to search again? (Y or N): n
```



# Checkpoint

1. 7.17 For what value in an array does the selection sort algorithm look first?

When the selection sort finds this value, what does it do with it?

2. 7.18 How many times will the selection sort swap the smallest value in an array with another value?
3. 7.19 Describe the difference between the sequential search and the binary search.
4. 7.20 On average, with an array of 20,000 elements, how many comparisons will the sequential search perform? (Assume the items being searched for are consistently found in the array.)
5. 7.21 If a sequential search is performed on an array, and it is known that some items are searched for more frequently than others, how can the contents of the array be reordered to improve the average performance of the search?

## 7.10 Two-Dimensional Arrays

### Concept:

A two-dimensional array is an array of arrays. It can be thought of as having rows and columns.

An array is useful for storing and working with a set of data. Sometimes, however, it's necessary to work with multiple sets of data. For example, in a grade-averaging program, a teacher might record all of one student's test scores in an array of doubles. If the teacher has 30 students, that means she'll need 30 arrays to record the scores for the entire class. Instead of defining 30 individual arrays, however, it would be better to define a two-dimensional array.

The arrays that you have studied so far are one-dimensional arrays. They are called *one-dimensional* because they can only hold one set of data. Two-dimensional arrays, which are sometimes called *2D arrays*, can hold multiple sets of data. Although a two-dimensional array is actually an array of arrays, it's best to think of it as having rows and columns of elements, as shown in [Figure 7-25](#). This figure shows an array of test scores, having three rows and four columns.

### Figure 7-25 Rows and columns

|       | Column 0 | Column 1 | Column 2 | Column 3 |
|-------|----------|----------|----------|----------|
| Row 0 |          |          |          |          |
| Row 1 |          |          |          |          |
| Row 2 |          |          |          |          |

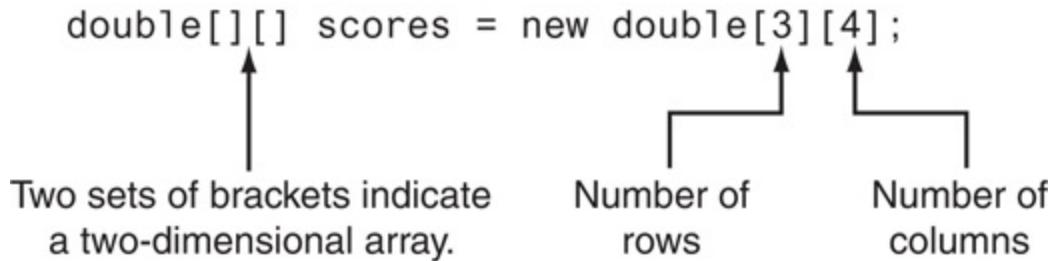
The array depicted in the figure has three rows (numbered 0 through 2) and four columns (numbered 0 through 3). There are a total of 12 elements in the array.

To declare a two-dimensional array, two sets of brackets and two size declarators are required: The first one is for the number of rows and the second one is for the number of columns. Here is an example declaration of a two-dimensional array with three rows and four columns:

```
double[][] scores = new double[3][4];
```

The two sets of brackets in the data type indicate that the `scores` variable will reference a two-dimensional array. The numbers 3 and 4 are size declarators. The first size declarator specifies the number of rows, and the second size declarator specifies the number of columns. Notice each size declarator is enclosed in its own set of brackets. This is illustrated in [Figure 7-26](#).

## Figure 7-26 Declaration of a two-dimensional array



[Figure 7-26 Full Alternative Text](#)

As with one-dimensional arrays, it is a common practice to use final variables as the size declarators for two-dimensional arrays. Here is an example:

```
final int ROWS = 3;
final int COLS = 4;
double[][] scores = new double[ROWS][COLS];
```

When processing the data in a two-dimensional array, each element has two subscripts: one for its row and another for its column. In the scores array, the elements in row 0 are referenced as:

```
scores[0][0]
scores[0][1]
scores[0][2]
scores[0][3]
```

The elements in row 1 are:

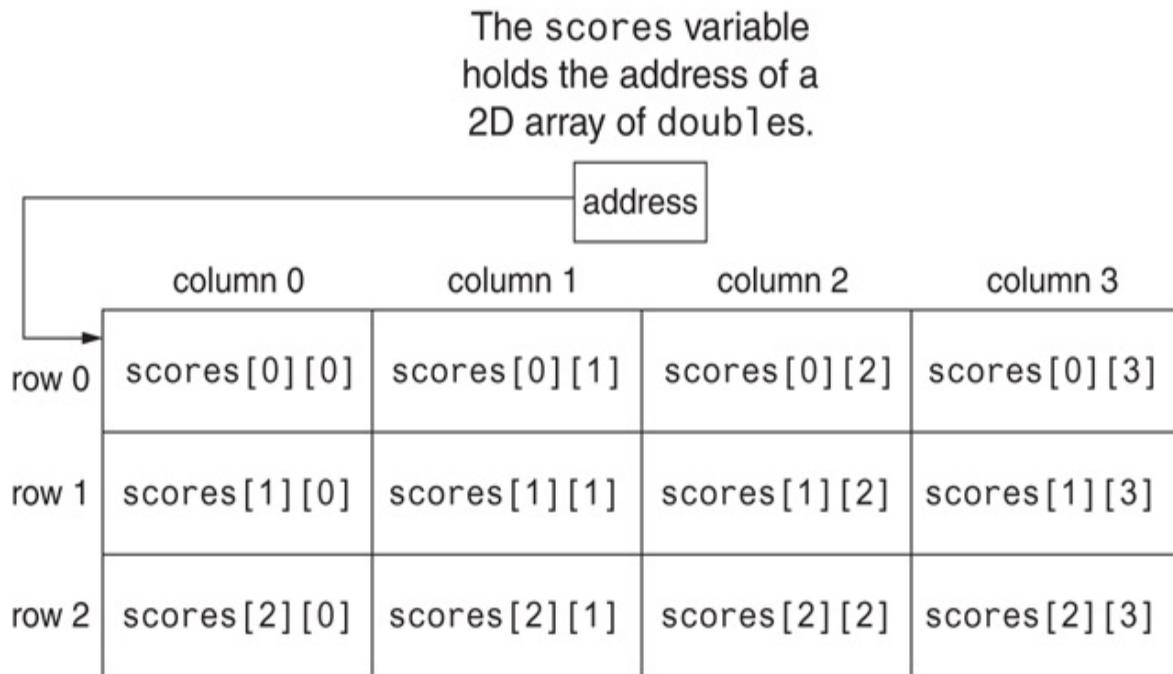
```
scores[1][0]
scores[1][1]
scores[1][2]
scores[1][3]
```

And the elements in row 2 are:

```
scores[2][0]
scores[2][1]
scores[2][2]
scores[2][3]
```

[Figure 7-27](#) illustrates the array with the subscripts shown for each element.

# Figure 7-27 Subscripts for each element of the scores array



[Figure 7-27 Full Alternative Text](#)

To access one of the elements in a two-dimensional array, you must use both subscripts. For example, the following statement stores the number 95 in scores[2][1]:

```
scores[2][1] = 95;
```

Programs that process two-dimensional arrays can do so with nested loops. For example, the following code prompts the user to enter a score, once for each element in the scores array:

```
final int ROWS = 3;
final int COLS = 4;
double[][] scores = new double[ROWS][COLS];
for (int row = 0; row < ROWS; row++)
{
 for (int col = 0; col < COLS; col++)
 {
```

```

 System.out.print("Enter a score: ");
 number = keyboard.nextDouble();
 scores[row][col] = number;
 }
}

```

And the following code displays all the elements in the scores array:

```

for (int row = 0; row < ROWS; row++)
{
 for (int col = 0; col < COLS; col++)
 {
 System.out.println(scores[row][col]);
 }
}

```

The program in [Code Listing 7-24](#) uses a two-dimensional array to store corporate sales data. The array has three rows (one for each division of the company) and four columns (one for each quarter).

## Code Listing 7-24 (CorpSales.java)

```

1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates a two-dimensional array.
5 */
6
7 public class CorpSales
8 {
9 public static void main(String[] args)
10 {
11 final int DIVS = 3; // Three divisions in the company
12 final int QTRS = 4; // Four quarters
13 double totalSales = 0.0; // Accumulator
14
15 // Create an array to hold the sales for each
16 // division, for each quarter.
17 double[][] sales = new double[DIVS][QTRS];
18
19 // Create a Scanner object for keyboard input.
20 Scanner keyboard = new Scanner(System.in);
21

```

```

22 // Display an introduction.
23 System.out.println("This program will calculate the " +
24 "total sales of");
25 System.out.println("all the company's divisions. " +
26 "Enter the following sales data:");
27
28 // Nested loops to fill the array with quarterly
29 // sales figures for each division.
30 for (int div = 0; div < DIVS; div++)
31 {
32 for (int qtr = 0; qtr < QTRS; qtr++)
33 {
34 System.out.printf("Division %d, Quarter %d: $",
35 (div + 1), (qtr + 1));
36 sales[div][qtr] = keyboard.nextDouble();
37 }
38 System.out.println(); // Print blank line.
39 }
40
41 // Nested loops to add all the elements of the array.
42 for (int div = 0; div < DIVS; div++)
43 {
44 for (int qtr = 0; qtr < QTRS; qtr++)
45 {
46 totalSales += sales[div][qtr];
47 }
48 }
49
50 // Display the total sales.
51 System.out.printf("Total company sales: $%,.2f\n",
52 totalSales);
53 }
54 }
```

## Program Output with Example Input Shown in Bold

This program will calculate the total sales of  
all the company's divisions. Enter the following sales data:

|                                          |                                      |
|------------------------------------------|--------------------------------------|
| Division 1, Quarter 1: <b>\$35698.77</b> | <input type="button" value="Enter"/> |
| Division 1, Quarter 2: <b>\$36148.63</b> | <input type="button" value="Enter"/> |
| Division 1, Quarter 3: <b>\$31258.95</b> | <input type="button" value="Enter"/> |
| Division 1, Quarter 4: <b>\$30864.12</b> | <input type="button" value="Enter"/> |

Division 2, Quarter 1: \$41289.64

Division 2, Quarter 2: \$43278.52

Division 2, Quarter 3: \$40927.18

Division 2, Quarter 4: \$42818.98

Division 3, Quarter 1: \$28914.56

Division 3, Quarter 2: \$27631.52

Division 3, Quarter 3: \$30596.64

Division 3, Quarter 4: \$29834.21

Total company sales: \$419,261.72

Look at the array declaration in line 17. As mentioned earlier, the array has three rows (one for each division) and four columns (one for each quarter) to store the company's sales data. The row subscripts are 0, 1, and 2, and the column subscripts are 0, 1, 2, and 3. [Figure 7-28](#) illustrates how the quarterly sales data is stored in the array.

## Figure 7-28 Division and quarter data stored in the sales array

The sales variable holds the address of a 2D array of doubles.

|       | column 0                                            | column 1                                            | column 2                                            | column 3                                            |
|-------|-----------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|
| row 0 | sales[0][0]<br>Holds data for division 1, quarter 1 | sales[0][1]<br>Holds data for division 1, quarter 2 | sales[0][2]<br>Holds data for division 1, quarter 3 | sales[0][3]<br>Holds data for division 1, quarter 4 |
| row 1 | sales[1][0]<br>Holds data for division 2, quarter 1 | sales[1][1]<br>Holds data for division 2, quarter 2 | sales[1][2]<br>Holds data for division 2, quarter 3 | sales[1][3]<br>Holds data for division 2, quarter 4 |
| row 2 | sales[2][0]<br>Holds data for division 3, quarter 1 | sales[2][1]<br>Holds data for division 3, quarter 2 | sales[2][2]<br>Holds data for division 3, quarter 3 | sales[2][3]<br>Holds data for division 3, quarter 4 |

[Figure 7-28 Full Alternative Text](#)

## Initializing a Two-Dimensional Array

When initializing a two-dimensional array, you enclose each row's initialization list in its own set of braces. Here is an example:

```
int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

As with one-dimensional arrays, you do not use the new key word when you provide an initialization list. Java automatically creates the array and fills its elements with the initialization values. In this example, the initialization values for row 0 are {1, 2, 3}, the initialization values for row 1 are {4, 5, 6},

and the initialization values for row 2 are {7, 8, 9}. So, this statement declares an array with three rows and three columns. The same statement could also be written as:

```
int[][] numbers = { {1, 2, 3},
 {4, 5, 6},
 {7, 8, 9} };
```

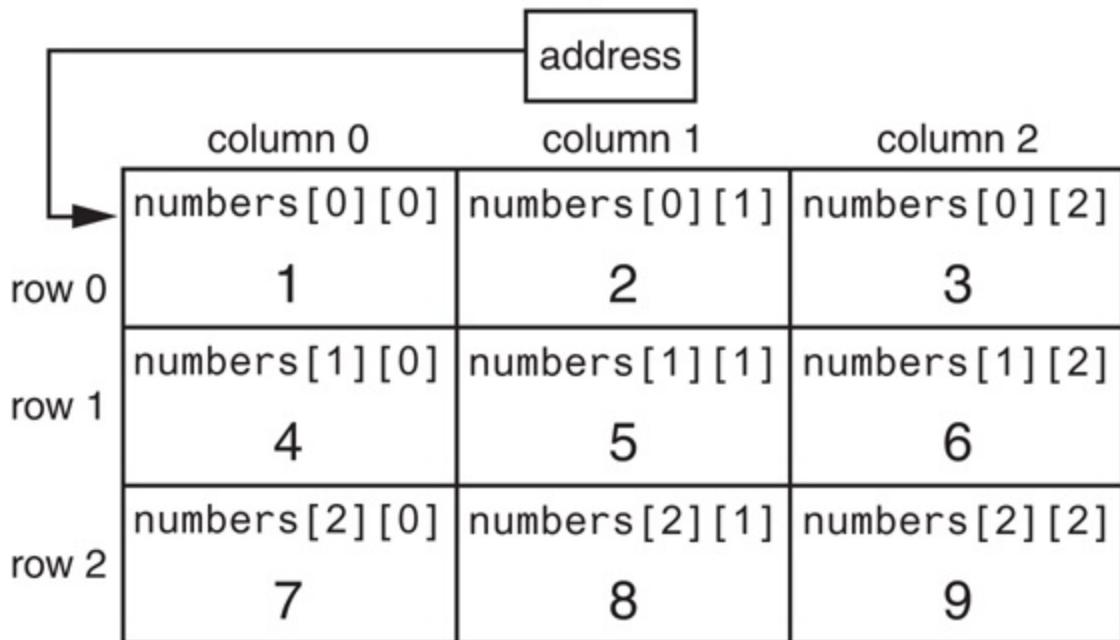
In either case, the values are assigned to the `numbers` array in the following manner:

```
numbers[0][0] is set to 1
numbers[0][1] is set to 2
numbers[0][2] is set to 3
numbers[1][0] is set to 4
numbers[1][1] is set to 5
numbers[1][2] is set to 6
numbers[2][0] is set to 7
numbers[2][1] is set to 8
numbers[2][2] is set to 9
```

[Figure 7-29](#) illustrates the array initialization.

## Figure 7-29 The numbers array

The numbers variable holds the address of a 2D array of ints.



[Figure 7-29 Full Alternative Text](#)

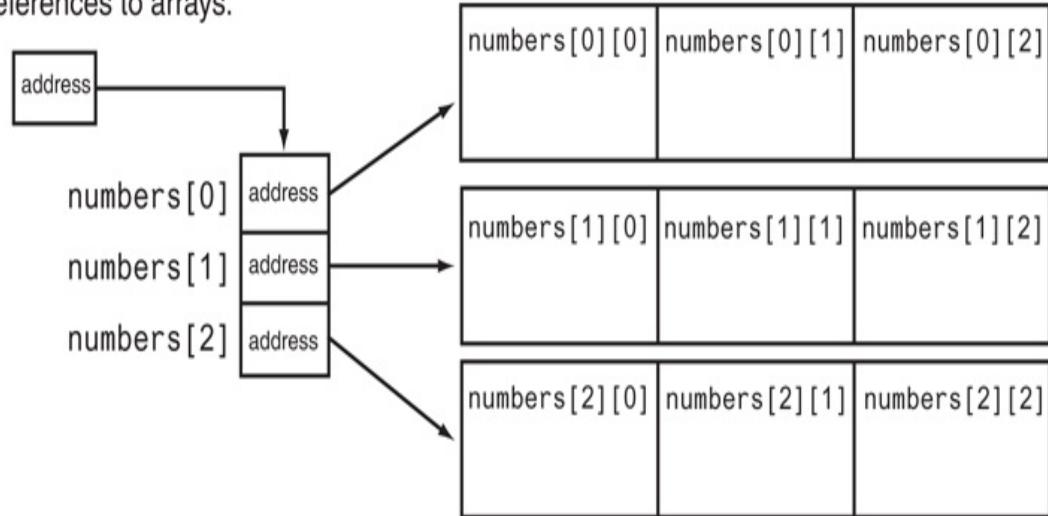
## The length Field in a Two-Dimensional Array

A one-dimensional array has a `length` field that holds the number of elements in the array. A two-dimensional array, however, has multiple `length` fields. It has a `length` field that holds the number of rows, then each row has a `length` field that holds the number of columns. This makes sense when you think of a two-dimensional array as an array of one-dimensional arrays. [Figure 7-29](#) shows the `numbers` array depicted in rows and columns. [Figure 7-30](#) shows another way of thinking of the `numbers` array: As an array of arrays.

## Figure 7-30 The numbers array

# is an array of arrays

The numbers variable holds the address of an array of references to arrays.



[Figure 7-30 Full Alternative Text](#)

As you can see from the figure, the numbers variable references a one-dimensional array with three elements. Each of the three elements is a reference to another one-dimensional array. The elements in the array referenced by numbers[0] are numbers[0][0], numbers[0][1], and numbers[0][2]. This pattern continues with numbers[1] and numbers[2]. The figure shows a total of four arrays. Each of the arrays in the figure has its own length field. The program in [Code Listing 7-25](#) uses these length fields to display the number of rows and columns in a two-dimensional array.

## Code Listing 7-25 (Lengths.java)

```
1 /**
2 * This program uses the length fields of a 2D array
3 * to display the number of rows and the number of
4 * columns in each row.
5 */
6
7 public class Lengths
```

```

8 {
9 public static void main(String[] args)
10 {
11 // Declare a 2D array with 3 rows
12 // and 4 columns.
13 int[][] numbers = { { 1, 2, 3, 4 },
14 { 5, 6, 7, 8 },
15 { 9, 10, 11, 12 } };
16
17 // Display the number of rows.
18 System.out.println("The number of rows is " +
19 numbers.length);
20
21 // Display the number of columns in each row.
22 for (int index = 0; index < numbers.length; index++)
23 {
24 System.out.println("The number of columns " +
25 "in row " + index + " is " +
26 numbers[index].length);
27 }
28 }
29 }
```

## **Program Output**

```

The number of rows is 3
The number of columns in row 0 is 4
The number of columns in row 1 is 4
The number of columns in row 2 is 4
```

# **Displaying All the Elements of a Two-Dimensional Array**

As you have seen in previous example programs, a pair of nested loops can be used to display all the elements of a two-dimensional array. For example, the following code creates the `numbers` array with three rows and four columns, then displays all the elements in the array:

```

int[][] numbers = { { 1, 2, 3, 4 },
 { 5, 6, 7, 8 },
 { 9, 10, 11, 12 } };
```

```
for (int row = 0; row < 3; row++)
{
 for (int col = 0; col < 4; col++)
 System.out.println(numbers[row][col]);
}
```

Although this code will display all of the elements, it is limited in the following way: The loops are specifically written to display an array with three rows and four columns. A better approach is to use the array's `length` fields for the upper limit of the subscripts in the loop test expressions. Here are the modified loops:

```
for (int row = 0; row < numbers.length; row++)
{
 for (int col = 0; col < numbers[row].length; col++)
 System.out.println(numbers[row][col]);
}
```

Let's take a closer look at the header for the outer loop:

```
for (int row = 0; row < numbers.length; row++)
```

This loop controls the subscript for the number array's rows. Because `numbers.length` holds the number of rows in the array, we have used it as the upper limit for the row subscripts. Here is the header for the inner loop:

```
for (int col = 0; col < numbers[row].length; col++)
```

This loop controls the subscript for the number array's columns. Because each row's `length` field holds the number of columns in the row, we have used it as the upper limit for the column subscripts. By using the `length` fields in algorithms that process two-dimensional arrays, you can write code that works with arrays of any number of rows and columns.

## Summing All the Elements of a Two-Dimensional Array

To sum all the elements of a two-dimensional array, you can use a pair of nested loops to add the contents of each element to an accumulator. The following code shows an example:

```
int[][] numbers = { { 1, 2, 3, 4 },
 { 5, 6, 7, 8 },
 { 9, 10, 11, 12 } };
int total; // Accumulator

total = 0; // Start the accumulator at 0.

// Sum the array elements.
for (int row = 0; row < numbers.length; row++)
{
 for (int col = 0; col < numbers[row].length; col++)
 total += numbers[row][col];
}

// Display the sum.
System.out.println("The total is " + total);
```

## Summing the Rows of a Two-Dimensional Array

Sometimes, you may need to calculate the sum of each row in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students. Each row in the array is a set of test scores for one student. To get the sum of a student's test scores (perhaps so that an average may be calculated), you use a loop to add all the elements in one row. The following code shows an example:

```
int[][] numbers = { { 1, 2, 3, 4 },
 { 5, 6, 7, 8 },
 { 9, 10, 11, 12 } };
int total; // Accumulator

for (int row = 0; row < numbers.length; row++)
{
 // Set the accumulator to 0.
 total = 0;
```

```

// Sum a row.
for (int col = 0; col < numbers[row].length; col++)
 total += numbers[row][col];

// Display the row's total.
System.out.println("Total of row " + row +
 " is " + total);
}

```

Notice that the `total` variable, which is used as an accumulator, is set to zero just before the inner loop executes. This is because the inner loop sums the elements of a row and stores the sum in `total`. Therefore, the `total` variable must be set to zero before each iteration of the inner loop.

## Summing the Columns of a Two-Dimensional Array

Sometimes, you may need to calculate the sum of each column in a two-dimensional array. For example, suppose a two-dimensional array is used to hold a set of test scores for a set of students, and you wish to calculate the class average for each of the test scores. To do this, you calculate the average of each column in the array. This is accomplished with a set of nested loops. The outer loop controls the column subscript, and the inner loop controls the row subscript. The inner loop calculates the sum of a column, which is stored in an accumulator. The following code demonstrates this:

```

int[][] numbers = { { 1, 2, 3, 4 },
 { 5, 6, 7, 8 },
 { 9, 10, 11, 12 } };
int total; // Accumulator

for (int col = 0; col < numbers[0].length; col++)
{
 // Set the accumulator to 0.
 total = 0;

 // Sum a column.
 for (int row = 0; row < numbers.length; row++)
 total += numbers[row][col];
}

```

```
// Display the column's total.
System.out.println("Total of column " + col +
 " is " + total);
}
```

## Passing Two-Dimensional Arrays to Methods

When a two-dimensional array is passed to a method, the parameter must be declared as a reference to a two-dimensional array. The following method header shows an example:

```
private static void showArray(int[][] array)
```

This method's parameter, `array`, is declared as a reference to a two-dimensional `int` array. Any two-dimensional `int` array can be passed as an argument to the method. [Code Listing 7-26](#) demonstrates two such methods.

## Code Listing 7-26 (Pass2Darray.java)

```
1 /**
2 * This class demonstrates methods that accept a two-
3 * dimensional array as an argument.
4 */
5
6 public class Pass2Darray
7 {
8 public static void main(String[] args)
9 {
10 // Create a 2D array of integers.
11 int[][] numbers = { { 1, 2, 3, 4 },
12 { 5, 6, 7, 8 },
13 { 9, 10, 11, 12 } };
14
15 System.out.println("Here are the values in " +
16 "the array.");
17 }
```

```
18 // Pass the numbers array to the showArray method.
19 // This will display the array's contents.
20 showArray(numbers);
21
22 // Display the sum of the array's values.
23 // Note the call to the arraySum method, with the
24 // array being passed as an argument.
25 System.out.println("The sum of the values is " +
26 arraySum(numbers));
27 }
28
29 /**
30 * The showArray method accepts a two-dimensional
31 * int array and displays its contents.
32 */
33
34 private static void showArray(int[][] array)
35 {
36 for (int row = 0; row < array.length; row++)
37 {
38 for (int col = 0; col < array[row].length; col++)
39 System.out.print(array[row][col] + " ");
40 System.out.println();
41 }
42 }
43
44 /**
45 * The arraySum method accepts a two-dimensional
46 * int array and returns the sum of its contents.
47 */
48
49 private static int arraySum(int[][] array)
50 {
51 int total = 0; // Accumulator
52
53 for (int row = 0; row < array.length; row++)
54 {
55 for (int col = 0; col < array[row].length; col++)
56 total += array[row][col];
57 }
58
59 return total;
60 }
61 }
```

## Program Output

```
Here are the values in the array.
```

```
1 2 3 4
5 6 7 8
9 10 11 12
```

```
The sum of the values is 78
```

## Ragged Arrays

Because the rows in a two-dimensional array are also arrays, each row can have its own length. When the rows of a two-dimensional array are of different lengths, the array is known as a *ragged array*. You create a ragged array by first creating a two-dimensional array with a specific number of rows, but no columns. Here is an example:

```
int[][] ragged = new int[4][];
```

This statement partially creates a two-dimensional array. The array can have four rows, but the rows have not yet been created. Next, you create the individual rows as shown in the following code.

```
ragged[0] = new int[3]; // Row 0 has 3 columns.
ragged[1] = new int[4]; // Row 1 has 4 columns.
ragged[2] = new int[5]; // Row 2 has 5 columns.
ragged[3] = new int[6]; // Row 3 has 6 columns.
```

This code creates the four rows. Row 0 has three columns, row 1 has four columns, row 2 has five columns, and row 3 has six columns. The following code displays the number of columns in each row:

```
for (int index = 0; index < ragged.length; index++)
{
 System.out.println("The number of columns " +
 "in row " + index + " is " + ragged[index].length);
}
```

This code will display the following output:

```
The number of columns in row 0 is 3
The number of columns in row 1 is 4
The number of columns in row 2 is 5
```

The number of columns in row 3 is 6

# 7.11 Arrays with Three or More Dimensions

## Concept:

Java does not limit the number of dimensions that an array may have. It is possible to create arrays with multiple dimensions, to model data that occurs in multiple sets.

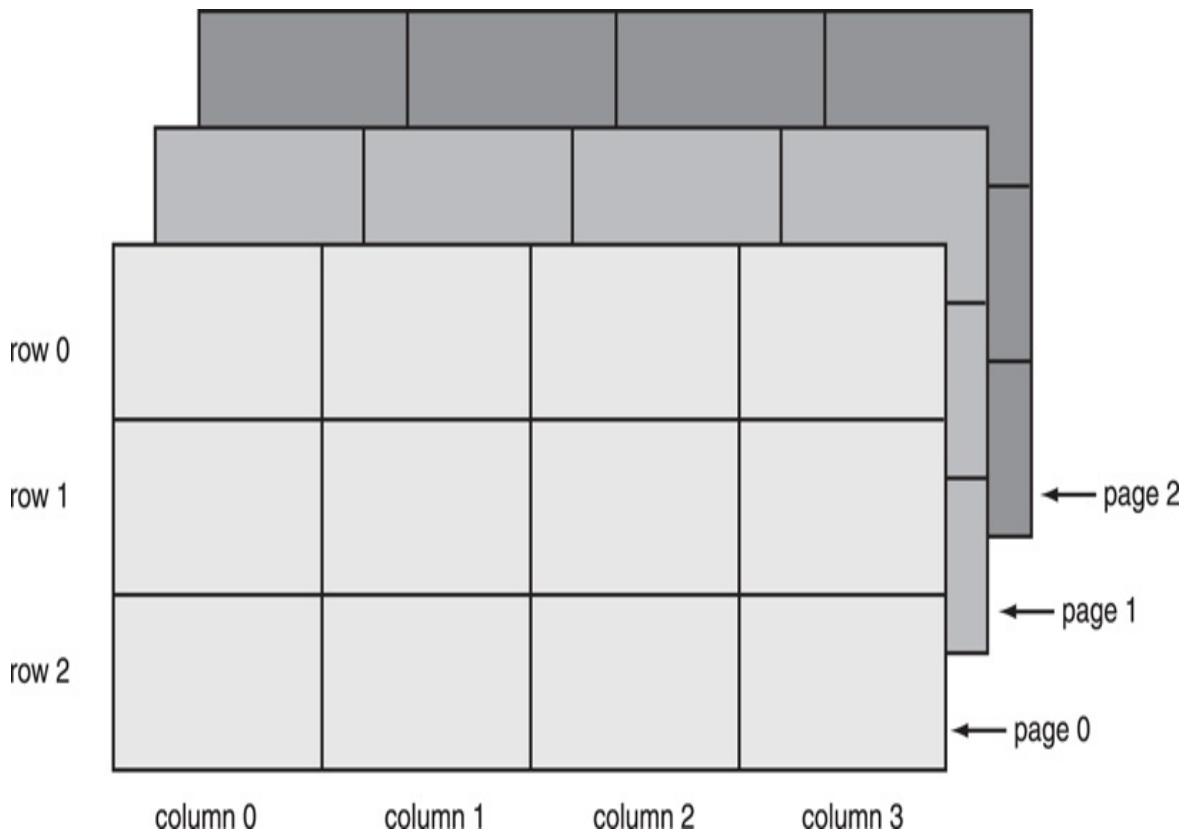
Java allows you to create arrays with virtually any number of dimensions. Here is an example of a three-dimensional array declaration:

```
double[][][] seats = new double[3][5][8];
```

This array can be thought of as three sets of five rows, with each row containing eight elements. The array might be used to store the prices of seats in an auditorium, where there are eight seats in a row, five rows in a section, and a total of three sections.

[Figure 7-31](#) illustrates the concept of a three-dimensional array as “pages” of two-dimensional arrays.

## Figure 7-31 A three-dimensional array



Arrays with more than three dimensions are difficult to visualize, but can be useful in some programming problems. For example, in a factory warehouse where cases of widgets are stacked on pallets, an array with four dimensions could be used to store a part number for each widget. The four subscripts of each element could represent the pallet number, case number, row number, and column number of each widget. Similarly, an array with five dimensions could be used if there were multiple warehouses.



## Checkpoint

1. 7.22 A video rental store keeps videos on 50 racks with 10 shelves each. Each shelf holds 25 videos. Declare a three-dimensional array large enough to represent the store's storage system.

# 7.12 Command-Line Arguments and Variable-Length Argument Lists

## Concept:

When you invoke a Java program from the operating system command line, you can specify arguments that are passed into the main method of the program. In addition, you can write a method that takes a variable number of arguments. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.

## Command-Line Arguments

Every program you have seen in this book, and every program you have written, uses a static `main` method with a header that looks like this:

```
public static void main(String[] args)
```

Inside the parentheses of the method header is the declaration of a parameter named `args`. This parameter is an array name. As its declaration indicates, it is used to reference an array of `Strings`. The array that is passed into the `args` parameter comes from the operating system command line. For example, look at [Code Listing 7-27](#).

## Code Listing 7-27 (`CommandLine.java`)

```
1 /**
2 * This program displays the arguments passed to
3 * it from the operating system command line.
4 */
5
6 public class CommandLine
7 {
8 public static void main(String [] args)
9 {
10 for (int i = 0; i < args.length; i++)
11 System.out.println(args[i]);
12 }
13 }
```

If this program is compiled and then executed with the command

```
java CommandLine How does this work?
```

its output will be

```
How
does
this
work?
```

Any items typed on the command line, separated by spaces, after the name of the class are considered one or more arguments to be passed into the `main` method. In the previous example, four arguments are passed into `args`. The word “How” is passed into `args[0]`, “does” is passed into `args[1]`, “this” is passed into `args[2]`, and “work?” is passed into `args[3]`. The for loop in `main` simply displays each argument.



## Note:

It is not required that the name of `main`’s parameter array be `args`. You can name it anything you wish. It is a standard convention, however, for the name `args` to be used.

# Variable-Length Argument Lists

Java provides a mechanism known as *variable-length argument lists*, which makes it possible to write a method that takes a variable number of arguments. In other words, you can write a method that accepts any number of arguments when it is called. When the method runs, it can determine the number of arguments that were passed to it and act accordingly.

For example, suppose we need to write a method named `sum` that can accept any number of `int` values, then return the sum of those values. We might call the method as shown here:

```
result = sum(10, 20);
```

Here we pass two arguments to the method: 10 and 20. After this code executes, the value 30 would be stored in the `result` variable. But the method does not have to accept two arguments each time it is called. We could call the method again with a different number of arguments, as shown here:

```
int firstVal = 1, secondVal = 2, thirdVal = 3, fourthVal = 4;
result = sum(firstVal, secondVal, thirdVal, fourthVal);
```

Here we pass four arguments to the method: `firstVal` (which is set to 1), `secondVal` (which is set to 2), `thirdVal` (which is set to 3), and `fourthVal` (which is set to 4). After this code executes, the value 10 would be stored in the `result` variable. Here's the code for the `sum` method:

```
public static int sum(int... numbers)
{
 int total = 0; // Accumulator

 // Add all the values in the numbers array.
 for (int val : numbers)
 total += val;

 // Return the total.
 return total;
}
```

Notice the declaration of the `numbers` parameter in the method header. The ellipsis (three periods) that follows the data type indicates that `numbers` is a special type of parameter known as a *vararg parameter*. A vararg parameter can take a variable number of arguments.

In fact, vararg parameters are actually arrays. In the `sum` method, the `numbers` parameter is an array of `ints`. All of the arguments passed to the `sum` method are stored in the elements of the `numbers` array. As you can see from the code, the method uses the enhanced `for` loop to step through the elements of the `numbers` array, adding up the values stored in its elements. (The `VarargsDemo1.java` program, available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis), demonstrates the `sum` method.)

You can also write a method to accept a variable number of object references as arguments. For example, the program in [Code Listing 7-28](#) shows a method that accepts a variable number of references to `InventoryItem` objects. The method returns the total of the objects' `units` fields.

## Code Listing 7-28 (`VarargsDemo2.java`)

```
1 /**
2 * This program demonstrates a method that accepts
3 * a variable number of arguments (varargs).
4 */
5
6 public class VarargsDemo2
7 {
8 public static void main(String[] args)
9 {
10 int total; // To hold the total units
11
12 // Create an InventoryItem object with 10 units.
13 InventoryItem item1 = new InventoryItem("Soap", 10);
14
15 // Create an InventoryItem object with 20 units.
16 InventoryItem item2 = new InventoryItem("Shampoo", 20);
17 }
```

```

18 // Create an InventoryItem object with 30 units.
19 InventoryItem item3 = new InventoryItem("Toothpaste", 30);
20
21 // Call the method with one argument.
22 total = totalUnits(item1);
23 System.out.println("Total: " + total);
24
25 // Call the method with two arguments.
26 total = totalUnits(item1, item2);
27 System.out.println("Total: " + total);
28
29 // Call the method with three arguments.
30 total = totalUnits(item1, item2, item3);
31 System.out.println("Total: " + total);
32 }
33
34 /**
35 * The totalUnits method takes a variable number
36 * of InventoryItem objects and returns the total
37 * of their units.
38 */
39
40 public static int totalUnits(InventoryItem... items)
41 {
42 int total = 0; // Accumulator
43
44 // Add all the values in the numbers array.
45 for (InventoryItem itemObject : items)
46 total += itemObject.getUnits();
47
48 // Return the total.
49 return total;
50 }
51 }
```

## Program Output

```
Total: 10
Total: 30
Total: 60
```

You can write a method to accept a mixture of fixed arguments and a variable-length argument list. For example, suppose you want to write a method named `courseAverage` that accepts the name of a course as a `String`, and a variable-length list of test scores as `doubles`. We could write the method header as:

```
public static void courseAverage(String course, double... scores)
```

This method has a regular `String` parameter named `course`, and a vararg parameter named `scores`. When we call this method, we always pass a `String` argument, then a list of `double` values. (This method is demonstrated in the program `VarargsDemo3.java`, which is available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).) Note that when a method accepts a mixture of fixed arguments and a variable-length argument list, the vararg parameter must be the last one declared.



## Note:

You can also pass an array to a vararg parameter. This is demonstrated in the program `VarargsDemo4.java`, which is available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

## 7.13 The ArrayList Class

### Concept:

`ArrayList` is a class in the Java API that is similar to an array and allows you to store objects. Unlike an array, an `ArrayList` object's size is automatically adjusted to accommodate the number of items being stored in it.

The Java API provides a class named `ArrayList`, which can be used for storing and retrieving objects. Once you create an `ArrayList` object, you can think of it as a container for holding other objects. An `ArrayList` object is similar to an array of objects, but offers many advantages over an array. Here are a few:

- An `ArrayList` object automatically expands as items are added to it.
- In addition to adding items to an `ArrayList`, you can remove items as well.
- An `ArrayList` object automatically shrinks as items are removed from it.

The `ArrayList` class is in the `java.util` package, so the following `import` statement is required:

```
import java.util.ArrayList;
```

### Creating and Using an ArrayList Object

Here is an example of how you create an `ArrayList` object:

```
ArrayList<String> nameList = new ArrayList<String>();
```

This statement creates a new `ArrayList` object and stores its address in the `nameList` variable. Notice in this example, the word `String` is written inside angled brackets, `<>`, immediately after the word `ArrayList`. This specifies that the `ArrayList` can hold `String` objects. If we try to store any other type of object in this `ArrayList`, an error occurs. (Later in this section, you will see an example that creates an `ArrayList` for holding other types of objects.)

To add items to the `ArrayList` object, you use the `add` method. For example, the following statements add a series of `String` objects to `nameList`:

```
nameList.add("James");
nameList.add("Catherine");
nameList.add("Bill");
```

After these statements execute, `nameList` will hold three references to `String` objects. The first will reference “James”, the second will reference “Catherine”, and the third will reference “Bill”.

The items stored in an `ArrayList` have a corresponding index. The index specifies the item’s location in the `ArrayList`, so it is much like an array subscript. The first item that is added to an `ArrayList` is stored at index 0. The next item that is added to the `ArrayList` is stored at index 1, and so forth. After the previously shown statements execute, “James” will be stored at index 0, “Catherine” will be stored at index 1, and “Bill” will be stored at index 2.

The `ArrayList` class has a `size` method that reports the number of items stored in an `ArrayList`. It returns the number of items as an `int`. For example, the following statement uses the method to display the number of items stored in `nameList`:

```
System.out.println("The ArrayList has " +
 nameList.size() +
 " objects stored in it.");
```

Assuming that `nameList` holds the strings “James”, “Catherine”, and “Bill”,

the following statement will display:

The ArrayList has 3 objects stored in it.

The ArrayList class's get method returns the item stored at a specific index. You pass the index as an argument to the method. For example, the following statement will display the item stored at index 1 of nameList:

```
System.out.println(nameList.get(1));
```

The program in [Code Listing 7-29](#) demonstrates the topics discussed so far.

## Code Listing 7-29 (ArrayListDemo1.java)

```
1 import java.util.ArrayList; // Needed for ArrayList class
2
3 /**
4 * This program demonstrates an ArrayList.
5 */
6
7 public class ArrayListDemo1
8 {
9 public static void main(String[] args)
10 {
11 // Create an ArrayList to hold some names.
12 ArrayList<String> nameList = new ArrayList<String>();
13
14 // Add some names to the ArrayList.
15 nameList.add("James");
16 nameList.add("Catherine");
17 nameList.add("Bill");
18
19 // Display the size of the ArrayList.
20 System.out.println("The ArrayList has " +
21 nameList.size() +
22 " objects stored in it.");
23
24 // Now display the items in nameList.
25 for (int index = 0; index < nameList.size(); index++)
26 System.out.println(nameList.get(index));
27 }
```

```
28 }
```

## Program Output

The ArrayList has 3 objects stored in it.

James  
Catherine  
Bill

Notice in line 25 the for loop uses the value returned from nameList's size method to control the number of times the loop iterates. This is to prevent a bounds checking error from occurring. The last item stored in an ArrayList will have an index that is 1 less than the size of the ArrayList. If you pass a value larger than this to the get method, an error will occur.

## Using the Enhanced for Loop with an ArrayList

Earlier in this chapter, you saw how the enhanced for loop can be used to iterate over each element in an array. You can also use the enhanced for loop to iterate over each item in an ArrayList. [Code Listing 7-30](#) demonstrates this. The enhanced for loop is used in lines 26 and 27 to display all the items stored in the ArrayList.

## Code Listing 7-30 (ArrayListDemo2.java)

```
1 import java.util.ArrayList; // Needed for ArrayList class
2
3 /**
4 * This program demonstrates how the enhanced for loop
5 * can be used with an ArrayList.
6 */
7
8 public class ArrayListDemo2
```

```
9 {
10 public static void main(String[] args)
11 {
12 // Create an ArrayList to hold some names.
13 ArrayList<String> nameList = new ArrayList<String>();
14
15 // Add some names to the ArrayList.
16 nameList.add("James");
17 nameList.add("Catherine");
18 nameList.add("Bill");
19
20 // Display the size of the ArrayList.
21 System.out.println("The ArrayList has " +
22 nameList.size() +
23 " objects stored in it.");
24
25 // Now display the items in nameList.
26 for (String name : nameList)
27 System.out.println(name);
28 }
29 }
```

## Program Output

The ArrayList has 3 objects stored in it.

James  
Catherine  
Bill

# The ArrayList Class's `toString` method

The `ArrayList` class has a `toString` method that returns a string representing all of the items stored in an `ArrayList` object. For example, suppose we have set up the `nameList` object as previously shown, with the strings “James”, “Catherine”, and “Bill”. We could use the following statement to display all of the names:

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

The contents of the `ArrayList` will be displayed in the following manner:

[James, Catherine, Bill]

# Removing an Item from an ArrayList

The `ArrayList` class has a `remove` method that removes an item at a specific index. You pass the index as an argument to the method. The program in [Code Listing 7-31](#) demonstrates this.

## Code Listing 7-31 (`ArrayListDemo3.java`)

```
1 import java.util.ArrayList; // Needed for ArrayList class
2
3 /**
4 * This program demonstrates an ArrayList.
5 */
6
7 public class ArrayListDemo3
8 {
9 public static void main(String[] args)
10 {
11 // Create an ArrayList to hold some names.
12 ArrayList<String> nameList = new ArrayList<String>();
13
14 // Add some names to the ArrayList.
15 nameList.add("James");
16 nameList.add("Catherine");
17 nameList.add("Bill");
18
19 // Display the items in nameList and their indices.
20 for (int index = 0; index < nameList.size(); index++)
21 {
22 System.out.println("Index: " + index + " Name: " +
23 nameList.get(index));
24 }
25
26 // Now remove the item at index 1.
27 nameList.remove(1);
```

```
28
29 System.out.println("The item at index 1 is removed. " +
30 "Here are the items now.");
31
32 // Display the items in nameList and their indices.
33 for (int index = 0; index < nameList.size(); index++)
34 {
35 System.out.println("Index: " + index + " Name: " +
36 nameList.get(index));
37 }
38 }
39 }
```

## Program Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
The item at index 1 is removed. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Bill
```

When the item at index 1 was removed (in line 27), the item that was previously stored at index 2 was shifted in position to index 1. When an item is removed from an `ArrayList`, the items that come after it are shifted downward in position to fill the empty space. This means that the index of each item after the removed item will be decreased by one.

Note an error will occur if you call the `remove` method with an invalid index.

## Inserting an Item

The `add` method, as previously shown, adds an item at the last position in an `ArrayList` object. The `ArrayList` class has an overloaded version of the `add` method that allows you to add an item at a specific index. This causes the item to be inserted into the `ArrayList` object at a specific position. The program in [Code Listing 7-32](#) demonstrates this.

## Code Listing 7-32

# (ArrayListDemo4.java)

```
1 import java.util.ArrayList; // Needed for ArrayList class
2
3 /**
4 * This program demonstrates inserting an item.
5 */
6
7 public class ArrayListDemo4
8 {
9 public static void main(String[] args)
10 {
11 // Create an ArrayList to hold some names.
12 ArrayList<String> nameList = new ArrayList<String>();
13
14 // Add some names to the ArrayList.
15 nameList.add("James");
16 nameList.add("Catherine");
17 nameList.add("Bill");
18
19 // Display the items in nameList and their indices.
20 for (int index = 0; index < nameList.size(); index++)
21 {
22 System.out.println("Index: " + index + " Name: " +
23 nameList.get(index));
24 }
25
26 // Now insert an item at index 1.
27 nameList.add(1, "Mary");
28
29 System.out.println("Mary was added at index 1. " +
30 "Here are the items now.");
31
32 // Display the items in nameList and their indices.
33 for (int index = 0; index < nameList.size(); index++)
34 {
35 System.out.println("Index: " + index + " Name: " +
36 nameList.get(index));
37 }
38 }
39 }
```

## Program Output

```
Index: 0 Name: James
Index: 1 Name: Catherine
Index: 2 Name: Bill
Mary was added at index 1. Here are the items now.
Index: 0 Name: James
Index: 1 Name: Mary
Index: 2 Name: Catherine
Index: 3 Name: Bill
```

When a new item was added at index 1 (in line 27), the item that was previously stored at index 1 was shifted in position to index 2. When an item is added at a specific index, the items that come after it are shifted upward in position to accommodate the new item. This means that the index of each item after the new item will be increased by one.

Note an error will occur if you call the add method with an invalid index.

## Replacing an Item

The `ArrayList` class's `set` method can be used to replace an item at a specific index with another item. For example, the following statement will replace the item currently at index 1 with the string "Jennifer":

```
nameList.set(1, "Jennifer");
```

This is demonstrated in the program *ArrayListDemo5.java*, which is available for download from the book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). Note an error will occur if you specify an invalid index.

## Capacity

Previously you learned that an `ArrayList` object's size is the number of items stored in the `ArrayList` object. When you add an item to the `ArrayList` object, its size increases by one, and when you remove an item from the `ArrayList` object, its size decreases by one.

An `ArrayList` object also has a *capacity*, which is the number of items it can store without having to increase its size. When an `ArrayList` object is first created, using the no-arg constructor, it has an initial capacity of 10 items. This means that it can hold up to 10 items without having to increase its size. When the 11th item is added, the `ArrayList` object must increase its size to accommodate the new item. You can specify a different starting capacity, if you desire, by passing an `int` argument to the `ArrayList` constructor. For example, the following statement creates an `ArrayList` object with an initial capacity of 100 items:

```
ArrayList<String> list = new ArrayList<String>(100);
```

All the examples we have looked at so far use `ArrayList` objects to hold `Strings`. You can create an `ArrayList` to hold any type of object. For example, the following statement creates an `ArrayList` that can hold `InventoryItem` objects:

```
ArrayList<InventoryItem> list = new ArrayList<InventoryItem>();
```

By specifying `InventoryItem` inside the angled brackets, we are declaring that the `ArrayList` can hold only `InventoryItem` objects. [Code Listing 7-33](#) demonstrates such an `ArrayList`.

## Code Listing 7-33 (`ArrayListDemo6.java`)

```
1 import java.util.ArrayList; // Needed for the ArrayList class
2
3 /**
4 * This program demonstrates how to use a cast operator
5 * with the ArrayList class's get method.
6 */
7
8 public class ArrayListDemo6
9 {
10 public static void main(String[] args)
11 {
12 // Create an ArrayList to hold InventoryItem objects.
13 ArrayList<InventoryItem> list = new ArrayList<InventoryItem>()
```

```

14
15 // Add three InventoryItem objects to the ArrayList.
16 list.add(new InventoryItem("Nuts", 100));
17 list.add(new InventoryItem("Bolts", 150));
18 list.add(new InventoryItem("Washers", 75));
19
20 // Display each item.
21 for (int index = 0; index < list.size(); index++)
22 {
23 InventoryItem item = (InventoryItem)list.get(index);
24 System.out.println("Item at index " + index +
25 "\nDescription: " + item.getDescription() +
26 "\nUnits: " + item.getUnits());
27 }
28 }
29 }
```

## Program Output

```

Item at index 0
Description: Nuts
Units: 100
Item at index 1
Description: Bolts
Units: 150
Item at index 2
Description: Washers
Units: 75
```

# Using the Diamond Operator for Type Inference (Java 7)

Beginning with Java 7, you can simplify the instantiation of an `ArrayList` by using the *diamond operator* (`<>`). For example, in this chapter you have seen several programs that create an `ArrayList` object with a statement such as this:

```
ArrayList<String> list = new ArrayList<String>();
```

Notice the data type (in this case, `String`) appears between the angled

brackets in two locations: first in the part that declares the reference variable, then again in the part that calls the `ArrayList` constructor. Beginning in Java 7, you are no longer required to write the data type in the part of the statement that calls the `ArrayList` constructor. Instead, you can simply write a set of empty angled brackets, as shown here:

```
ArrayList<String> list = new ArrayList<>();
```

The set of empty angled brackets (`<>`), the diamond operator, causes the compiler to infer the required data type from the reference variable declaration. Here is another example:

```
ArrayList<InventoryItem> list = new ArrayList<>();
```

This creates an `ArrayList` that can hold `InventoryItem` objects. Keep in mind, type inference was introduced in Java 7. If you are using a previous version of the Java language, you will have to use the more lengthy form of the declaration statement to create an `ArrayList`.



## Checkpoint

1. 7.23 What `import` statement must you include in your code to use the `ArrayList` class?
2. 7.24 Write a statement that creates an `ArrayList` object and assigns its address to a variable named `frogs`.
3. 7.25 Write a statement that creates an `ArrayList` object and assigns its address to a variable named `lizards`. The `ArrayList` should be able to store `String` objects only.
4. 7.26 How do you add items to an `ArrayList` object?
5. 7.27 How do you remove an item from an `ArrayList` object?
6. 7.28 How do you retrieve a specific item from an `ArrayList` object?

7. 7.29 How do you insert an item at a specific location in an `ArrayList` object?
8. 7.30 How do you determine an `ArrayList` object's size?
9. 7.31 What is the difference between an `ArrayList` object's size and its capacity?

## 7.14 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- Using an invalid subscript. Java does not allow you to use a subscript value that is outside the range of valid subscripts for an array.
- Confusing the contents of an integer array element with the element's subscript. An element's subscript and the value stored in the element are not the same thing. The subscript identifies an element, which holds a value.
- Causing an off-by-one error. When processing arrays, the subscripts start at 0 and end at 1 less than the number of elements in the array. Off-by-one errors are commonly caused when a loop uses an initial subscript of 1 and/or uses a maximum subscript that is equal to the number of elements in the array.
- Using the = operator to copy an array. Assigning one array reference variable to another with the = operator merely copies the address in one variable to the other. To copy an array, you should copy the individual elements of one array to another.
- Using the == operator to compare two arrays. You cannot use the == operator to compare two array reference variables and determine whether the arrays are equal. When you use the == operator with reference variables, the operator compares the memory addresses that the variables contain, not the contents of the objects referenced by the variables.
- Reversing the row and column subscripts when processing a two-dimensional array. When thinking of a two-dimensional array as having rows and columns, the first subscript accesses a row and the second subscript accesses a column. If you reverse these subscripts, you will

access the wrong element.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. This indicates in an array declaration the number of elements that an array is to have.
  1. subscript
  2. size declarator
  3. element sum
  4. reference variable
2. Each element of an array is accessed by a number known as a(n) .
  1. subscript
  2. size declarator
  3. address
  4. specifier
3. The first subscript in an array is always .
  1. 1
  2. 0
  3. -1
  4. 1 less than the number of elements

4. The last subscript in an array is always
  1. 100
  2. 0
  3. -1
  4. 1 less than the number of elements
5. Array bounds checking happens
  1. when the program is compiled
  2. when the program is saved
  3. when the program runs
  4. when the program is loaded into memory
6. This array field holds the number of elements that the array has.
  1. size
  2. elements
  3. length
  4. width
7. This search algorithm steps through an array, comparing each item with the search value.
  1. binary search
  2. sequential search
  3. selection search
  4. iterative search

8. This search algorithm repeatedly divides the portion of an array being searched in half.
  1. binary search
  2. sequential search
  3. selection search
  4. iterative search
9. This is the *maximum* number of comparisons performed by the sequential search on an array of  $N$  elements (assuming the search values are consistently found).
  1.  $2N$
  2.  $N$
  3.  $N^2$
  4.  $N/2$
10. When initializing a two-dimensional array, you enclose each row's initialization list in these.
  1. braces
  2. parentheses
  3. brackets
  4. quotation marks
11. To store an item in an `ArrayList` object, use this method.
  1. `store`
  2. `insert`

- 3. add
  - 4. get
- 12. To insert an item at a specific location in an `ArrayList` object, use this method.
  - 1. store
  - 2. insert
  - 3. add
  - 4. get
- 13. To delete an item from an `ArrayList` object, use this method.
  - 1. remove
  - 2. delete
  - 3. erase
  - 4. get
- 14. To determine the number of items stored in an `ArrayList` object, use this method.
  - 1. size
  - 2. capacity
  - 3. items
  - 4. length
- 15. **True or False:** Java does not allow a statement to use a subscript that is outside the range of valid subscripts for an array.

16. **True or False:** An array's size declarator can be a negative integer expression.
17. **True or False:** Both of the following declarations are legal and equivalent:

```
int[] numbers;
int numbers[];
```
18. True or False: The subscript of the last element in a one-dimensional array is one less than the total number of elements in the array.
19. True or False: The values in an initialization list are stored in the array in the order they appear in the list.
20. True or False: The Java compiler does not display an error message when it processes a statement that uses an invalid subscript.
21. True or False: When an array is passed to a method, the method has access to the original array.
22. True or False: The first size declarator in the declaration of a two-dimensional array represents the number of columns. The second size declarator represents the number of rows.
23. True or False: A two-dimensional array has multiple `length` fields.
24. True or False: An `ArrayList` automatically expands in size to accommodate the items stored in it.

## Find the Error

1. `int[] collection = new int[-20];`
2. `int[] hours = 8, 12, 16;`
3. `int[] table = new int[10];`

```
Scanner keyboard = new Scanner(System.in);
for (int x = 1; x <= 10; x++)
{
 System.out.print("Enter the next value: ");
 table[x] = keyboard.nextInt();
}
```

4. String[] names = { "George", "Susan" };
 int totalLength = 0;
 for (int i = 0; i < names.length(); i++)
 totalLength += names[i].length;
  
5. String[] words = { "Hello", "Goodbye" };
 System.out.println(words.toUpperCase());

## Algorithm Workbench

1. The variable `names` references an integer array with 20 elements. Write a `for` loop that prints each element of the array.
  
2. The variables `numberArray1` and `numberArray2` reference arrays that each have 100 elements. Write code that copies the values in `numberArray1` to `numberArray2`.
  
3.
  1. Write a statement that declares a `String` array initialized with the following strings: “Einstein”, “Newton”, “Copernicus”, and “Kepler”.
  
  2. Write a loop that displays the contents of each element in the array you declared in Part a.
  
  3. Write code that displays the total length of all the strings in the array you declared in Part a.
  
4. In a program, you need to store the populations of 12 countries.
  1. Define two arrays that may be used in parallel to store the names of the countries and their populations.

2. Write a loop that uses these arrays to print each country's name and its population.
5. In a program, you need to store the identification numbers of 10 employees (as integers) and their weekly gross pay (as double values).
  1. Define two arrays that may be used in parallel to store the 10 employee identification numbers and gross pay amounts.
  2. Write a loop that uses these arrays to print each of the employees' identification number and weekly gross pay.
6. Declare and create a two-dimensional int array named grades. It should have 30 rows and 10 columns.
7. Write code that calculates the average of all the elements in the grades array that you declared in Question 6.
8. Look at the following array declaration:

```
int[][] numberArray = new int[9][11];
```
1. Write a statement that assigns 145 to the first column of the first row of this array.
2. Write a statement that assigns 18 to the last column of the last row of this array.
9. The values variable references a two-dimensional double array with 10 rows and 20 columns. Write code that sums all the elements in the array and stores the sum in the variable total.
10. An application uses a two-dimensional array declared as follows:

```
int[][] days = new int[29][5];
```

1. Write code that sums each row in the array and displays the results.
2. Write code that sums each column in the array and displays the

results.

11. Write code that creates an `ArrayList` object that can hold `String` objects. Add the names of three cars to the `ArrayList`, then display the contents of the `ArrayList`.

## Short Answer

1. What is the difference between an array size declarator and a subscript?
2. Look at the following array definition:

```
int[] values = new int[10];
```

1. How many elements does the array have?
  2. What is the subscript of the first element in the array?
  3. What is the subscript of the last element in the array?
3. In the following array definition:

```
int[] values = { 4, 7, 6, 8, 2 };
```

what does each of the following code segments display?

```
System.out.println(values[4]); a.
```

```
x = values[2] + values[3]; b.
System.out.println(x);
```

```
x = ++values[1]; c.
System.out.println(x);
```

4. How do you define an array without providing a size declarator?

- Assuming that `array1` and `array2` are both array reference variables, why is it not possible to assign the contents of the array referenced by `array2` to the array referenced by `array1` with the following statement?

```
array1 = array2;
```

- The following statement creates an `InventoryItem` array:

```
InventoryItem[] items = new InventoryItem[10];
```

Is it okay or not okay to execute the following statements?

```
items[0].setDescription("Hammer");
```

```
items[0].setUnits(10);
```

- If a sequential search method is searching for a value that is stored in the last element of a 10,000-element array, how many elements will the search code have to examine to locate the value?
- Look at the following array definition:

```
double[][] sales = new double[8][10];
```

- How many rows does the array have?
- How many columns does the array have?
- How many elements does the array have?
- Write a statement that stores a number in the last column of the last row in the array.

# Programming Challenges

## 1. Rainfall Class

Write a `RainFall` class that stores the total rainfall for each of 12 months into an array of doubles. The program should have methods that return the following:

- total rainfall for the year
- the average monthly rainfall
- the month with the most rain
- the month with the least rain

Demonstrate the class in a complete program.

*Input Validation: Do not accept negative numbers for monthly rainfall figures.*

## 2. Payroll Class

Write a `Payroll` class that uses the following arrays as fields:

- `employeeId`. An array of seven integers to hold employee identification numbers. The array should be initialized with the following numbers:

```
5658845 4520125 7895122 8777541
8451277 1302850 7580489
```

- `hours`. An array of seven integers to hold the number of hours worked by each employee
- `payRate`. An array of seven doubles to hold each employee's hourly pay rate

- wages . An array of seven doubles to hold each employee’s gross wages

The class should relate the data in each array through the subscripts. For example, the number in element 0 of the hours array should be the number of hours worked by the employee whose identification number is stored in element 0 of the employeeId array. That same employee’s pay rate should be stored in element 0 of the payRate array.

In addition to the appropriate accessor and mutator methods, the class should have a method that accepts an employee’s identification number as an argument and returns the gross pay for that employee.

Demonstrate the class in a complete program that displays each employee number and asks the user to enter that employee’s hours and pay rate. It should then display each employee’s identification number and gross wages.

*Input Validation: Do not accept negative values for hours or numbers less than 6.00 for pay rate.*

### 3. Charge Account Validation

Create a class with a method that accepts a charge account number as its argument. The method should determine whether the number is valid by comparing it to the following list of valid charge account numbers:

|         |         |         |         |         |        |
|---------|---------|---------|---------|---------|--------|
| 5658845 | 4520125 | 7895122 | 8777541 | 8451277 | 130285 |
| 8080152 | 4562555 | 5552012 | 5050552 | 7825877 | 125025 |
| 1005231 | 6545231 | 3852085 | 7576651 | 7881200 | 458100 |

These numbers should be stored in an array. Use either a sequential search or a binary search to locate the number passed as an argument. If the number is in the array, the method should return true, indicating the number is valid. If the number is not in the array, the method should return false, indicating the number is invalid.

Write a program that tests the class by asking the user to enter a charge account number. The program should display a message indicating

whether the number is valid or invalid.



#### VideoNote The Charge Account Validation Problem

#### 4. Larger Than $n$

In a program, write a method that accepts two arguments: an array and a number  $n$ . Assume the array contains integers. The method should display all of the numbers in the array that are greater than the number  $n$ .

#### 5. Charge Account Modification

Modify the charge account validation class you wrote for Programming Challenge 3 so it reads the list of valid charge account numbers from a file. Use Notepad or another text editor to create the file.

#### 6. Driver's License Exam

The local driver's license office has asked you to write a program that grades the written portion of the driver's license exam. The exam has 20 multiple-choice questions. Here are the correct answers:

1. B
2. D
3. A
4. A
5. C
6. A
7. B
8. A
9. C
10. D
11. B
12. C
13. D
14. A
15. D
16. C
17. C
18. B
19. D
20. A

A student must correctly answer 15 of the 20 questions to pass the exam.

Write a class named `DriverExam` that holds the correct answers to the exam in an array field. The class should also have an array field that holds the student's answers. The class should have the following methods:

- `passed`. Returns `true` if the student passed the exam, or `false` if

the student failed

- `totalCorrect`. Returns the total number of correctly answered questions
- `totalIncorrect`. Returns the total number of incorrectly answered questions
- `questionsMissed`: An `int` array containing the question numbers that the student missed

Demonstrate the class in a complete program that asks the user to enter a student’s answers, then displays the results returned from the `DriverExam` class’s methods.

*Input Validation: Only accept the letters A, B, C, or D as answers.*

## 7. Magic 8 Ball

Write a program that simulates a Magic 8 Ball, which is a fortune-telling toy that displays a random response to a yes or no question. In the student sample programs for this book, you will find a text file named `8_ball_responses.txt`. The file contains 12 responses, such as “I don’t think so”, “Yes, of course!”, “I’m not sure”, and so forth. The program should read the responses from the file into an array or `ArrayList` object. It should prompt the user to ask a question, and then display one of the responses, randomly selected from the array or `ArrayList` object. The program should repeat until the user is ready to quit.

***Contents of `8_ball_responses.txt`:***

Yes, of course!  
Without a doubt, yes.  
You can count on it.  
For sure!  
Ask me later.  
I'm not sure.  
I can't tell you right now.  
I'll tell you after my nap.  
No way!

I don't think so.  
Without a doubt, no.  
The answer is clearly NO.

## 8. Grade Book

A teacher has five students who have taken four tests. The teacher uses the following grading scale to assign a letter grade to a student, based on the average of his or her four test scores:

### **Test Score Letter Grade**

|        |   |
|--------|---|
| 90–100 | A |
| 80–89  | B |
| 70–79  | C |
| 60–69  | D |
| 0–59   | F |

Write a class that uses a `String` array (or an `ArrayList` object) to hold the five students' names, an array of five characters to hold the five students' letter grades, and five arrays of four doubles each to hold each student's set of test scores. The class should have methods that return a specific student's name, average test score, and a letter grade based on the average.

Demonstrate the class in a program that allows the user to enter each student's name and his or her four test scores. It should then display each student's average test score and letter grade.

***Input Validation:** Do not accept test scores less than zero or greater than 100.*

## 9. Grade Book Modification

Modify the grade book application in Programming Challenge 7 so it drops each student's lowest score when determining the test score averages and letter grades.

## 10. Average Steps Taken

A Personal Fitness Tracker is a wearable device that tracks your physical activity, calories burned, heart rate, sleeping patterns, and so on. One common physical activity most of these devices track is the number of steps you take each day.

If you have downloaded this book's source code from the companion website, you will find a file named steps.txt in the [Chapter 7](#) folder. (The companion website can be found at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).) The steps.txt file contains the number of steps a person has taken each day for a year. There are 365 lines in the file, and each line contains the number of steps taken during a day. (The first line is the number of steps taken on January 1st, the second line is the number of steps taken on January 2nd, and so forth.) Write a program that reads the file, then displays the average number of steps taken for each month. (The data is from a year that was not a leap year, so February has 28 days.)

## 11. ArrayOperations Class

Write a class name `ArrayOperations` with the following static methods:

- `getTotal`. This method should accept a one-dimensional array as its argument and return the total of the values in the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getAverage`. This method should accept a one-dimensional array as its argument and return the average of the values in the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getHighest`. This method should accept a one-dimensional array as its argument and return the highest value in the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getLowest`. This method should accept a one-dimensional array as its argument and return the lowest value in the array. Write overloaded versions of this method that work with `int`, `float`,

`double`, and `long` arrays.

Demonstrate the class in a complete program with test data stored in arrays of various data types.

## 12. 1994 Gas Prices

In the student sample programs for this book, you will find a text file named `1994_Weekly_Gas_Averages.txt`. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1, line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file into an array or an `ArrayList` object. The program should do the following:

- Display the lowest average price of the year, along with the week number for that price, and the name of the month in which it occurred.
- Display the highest average price of the year, along with the week number for that price, and the name of the month in which it occurred.
- Display the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.)

## 13. Sorted List of 1994 Gas Prices



### Note:

This assignment can be done either as an enhancement to the program that you wrote for Programming Challenge 12, or as a stand-alone program.

In the student sample programs for this book, you will find a text file

named `1994_Weekly_Gas_Averages.txt`. The file contains the average gas price for each week in the year 1994. (There are 52 lines in the file. Line 1 contains the average price for week 1, line 2 contains the average price for week 2, and so forth.) Write a program that reads the gas prices from the file, and calculates the average gas price for each month. (To get the average price for a given month, calculate the average of the average weekly prices for that month.) Then, the program should create another file that lists the names of the months, along with each month's average gas price, sorted from lowest to highest.

#### 14. Name Search

If you have downloaded this book's source code (the companion website is available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)), you will find the following files in the [Chapter 7](#) folder:

- `GirlNames.txt`—This file contains a list of the 200 most popular names given to girls born in the United States from the year 2000 to 2009.
- `BoyNames.txt`—This file contains a list of the 200 most popular names given to boys born in the United States from the year 2000 to 2009.

Write a program that reads the contents of the two files into two separate arrays, or `ArrayLists`. The user should be able to enter a boy's name, a girl's name, or both, and the application will display messages indicating whether the names were among the most popular.

#### 15. Population Data

If you have downloaded this book's source code (the companion website is available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)), you will find a file named `USPopulation.txt` in the [Chapter 7](#) folder. The file contains the midyear population of the United States, in thousands, during the years 1950 to 1990. The first line in the file contains the population for 1950, the second line contains the population for 1951, and so forth.

Write a program that reads the file's contents into an array, or an `ArrayList`. The program should display the following data:

- The average annual change in population during the time period
- The year with the greatest increase in population during the time period
- The year with the smallest increase in population during the time period

## 16. World Series Champions

If you have downloaded this book's source code (the companion website is available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)), you will find a file named `WorldSeriesWinners.txt`. This file contains a chronological list of the World Series winning teams from 1903 to 2009. (The first line in the file is the name of the team that won in 1903, and the last line is the name of the team that won in 2009. Note that the World Series was not played in 1904 and 1994, so the file contains no entries for those years.)

Write a program that lets the user enter the name of a team, then displays the number of times that team has won the World Series in the time period from 1903 to 2009.



Read the contents of the `WorldSeriesWinners.txt` file into an array, or an `ArrayList`. When the user enters the name of a team, the program should step through the array or `ArrayList` counting the number of times the selected team appears.

## 17. `2DArrayOperations` Class

Write a class named `2DArrayOperations` with the following static methods:

- `getTotal`. This method should accept a two-dimensional array as its argument and return the total of all the values in the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getAverage`. This method should accept a two-dimensional array as its argument and return the average of all the values in the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getRowTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the total of the values in the specified row. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getColumnTotal`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a column in the array. The method should return the total of the values in the specified column. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getHighestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the highest value in the specified row of the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.
- `getLowestInRow`. This method should accept a two-dimensional array as its first argument and an integer as its second argument. The second argument should be the subscript of a row in the array. The method should return the lowest value in the specified row of the array. Write overloaded versions of this method that work with `int`, `float`, `double`, and `long` arrays.

Demonstrate the class in a complete program with test data stored in two-dimensional arrays of various data types.

## 18. Search Benchmarks

Modify the `sequentialSearch` and `binarySearch` methods presented in this chapter so they keep a count of and display on the screen the number of comparisons they make before finding the value they are searching for. Then write a program that has an array of at least 20 integers. It should call the `sequentialSearch` method to locate at least five of the values. Then it should call the `binarySearch` method to locate the same values. On average, which method makes the fewest comparisons?

## 19. Phone Book ArrayList

Write a class named `PhoneBookEntry` that has fields for a person's name and phone number. The class should have a constructor and appropriate accessor and mutator methods. Then write a program that creates at least five `PhoneBookEntry` objects and stores them in an `ArrayList`. Use a loop to display the contents of each object in the `ArrayList`.

## 20. Trivia Game

In this programming challenge, you will create a simple trivia game for two players. The program will work like this:

- Starting with player 1, each player gets a turn at answering 5 trivia questions. (There are a total of 10 questions, 5 for each player.) When a question is displayed, four possible answers are also displayed. Only one of the answers is correct, and if the player selects the correct answer, he or she earns a point.
- After answers have been selected for all the questions, the program displays the number of points earned by each player, and declares the player with the highest number of points the winner.

You are to design a `Question` class to hold the data for a trivia question.

The Question class should have fields for the following data:

- A trivia question
- Possible answer #1
- Possible answer #2
- Possible answer #3
- Possible answer #4
- The number of the correct answer (1, 2, 3, or 4)

The Question class should have appropriate constructor(s), accessor, and mutator methods.

The program should create an array of 10 Question objects, one for each trivia question. (If you prefer, you can use an ArrayList instead of an array.) Make up your own trivia questions on the subject or subjects of your choice for the objects.

## 21. Lo Shu Magic Square

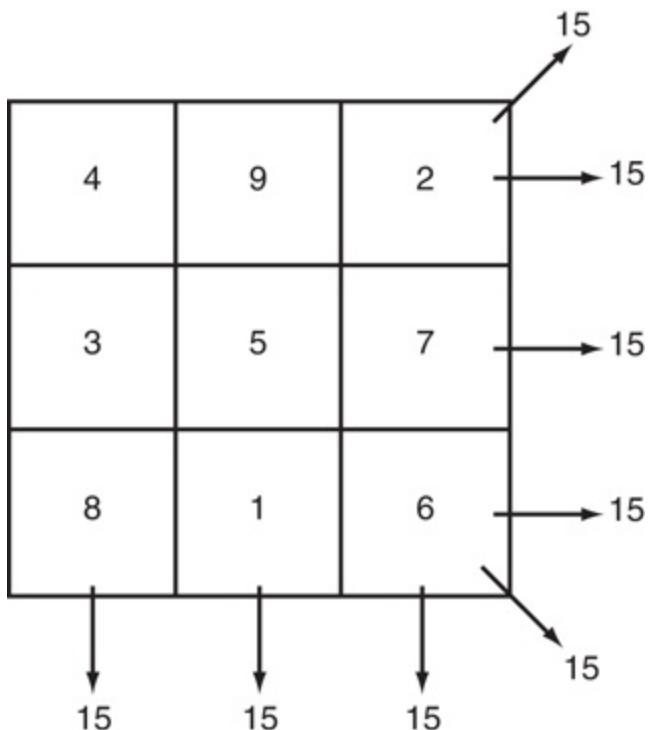
The Lo Shu Magic Square is a grid with 3 rows and 3 columns shown in [Figure 7-32](#). The Lo Shu Magic Square has the following properties:

- The grid contains the numbers 1 through 9 exactly.
- The sum of each row, each column, and each diagonal all add up to the same number. This is shown in [Figure 7-33](#).

# Figure 7-32 Lo Shu Magic Square

|   |   |   |
|---|---|---|
| 4 | 9 | 2 |
| 3 | 5 | 7 |
| 8 | 1 | 6 |

**Figure 7-33 Row, column, and diagonal sums in the Lo Shu Magic Square**



[Figure 7-33 Full Alternative Text](#)

In a program, you can simulate a magic square using a two-dimensional array. Write a method that accepts a two-dimensional array as an argument and determines whether the array is a Lo Shu Magic Square. Test the method in a program.

# Chapter 8 Text Processing and Wrapper Classes

## Topics

1. [8.1 Introduction to Wrapper Classes](#)
2. [8.2 Character Testing and Conversion with the Character Class](#)
3. [8.3 More about String Objects](#)
4. [8.4 The StringBuilder Class](#)
5. [8.5 Tokenizing Strings](#)
6. [8.6 Wrapper Classes for the Numeric Data Types](#)
7. [8.7 Focus on Problem Solving: The TestScoreReader Class](#)
8. [8.8 Common Errors to Avoid](#)

# 8.1 Introduction to Wrapper Classes

## Concept:

Java provides wrapper classes for the primitive data types. The wrapper class for a given primitive type contains not only a value of that type, but also methods that perform operations related to the type.

Recall from [Chapter 2](#) that the primitive data types are called “primitive” because they are not created from classes. Instead of instantiating objects, you create variables from the primitive data types, and variables do not have attributes or methods. They are designed simply to hold a single value in memory.

Java also provides wrapper classes for all of the primitive data types. A *wrapper class* is a class that is “wrapped around” a primitive data type, and allows you to create objects instead of variables. In addition, these wrapper classes provide methods that perform useful operations on primitive values.

Although these wrapper classes can be used to create objects instead of variables, few programmers use them that way. One reason is because the wrapper classes are immutable, which means that once you create an object, you cannot change the object’s value. Another reason is because they are not as easy to use as variables for simple operations. For example, to get the value stored in an object, you must call a method, whereas variables can be used directly in assignment statements, passed as arguments to the `print` and `println` methods, and so forth.

Although it is not normally useful to create objects from the wrapper classes, they do provide static methods that are very useful. We will examine several of Java’s wrapper classes in this chapter. We begin by looking at the `Character` class, which is the wrapper class for the `char` data type.

# 8.2 Character Testing and Conversion with the Character Class

## Concept:

The Character class is a wrapper class for the char data type. It provides numerous methods for testing and converting character data.

The Character class is part of the `java.lang` package, so no `import` statement is necessary to use this class. The class provides several static methods for testing the value of a `char` variable. Some of these methods are listed in [Table 8-1](#). Each of the methods accepts a single `char` argument and returns a boolean value.

**Table 8-1 Some static Character class methods for testing char values**

| Method                                 | Description                                                                                                                               |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean isDigit(char ch)</code>  | Returns <code>true</code> if the argument passed into <code>ch</code> is a digit from 0 through 9. Otherwise returns <code>false</code> . |
| <code>boolean isLetter(char ch)</code> | Returns <code>true</code> if the argument passed into <code>ch</code> is an alphabetic letter. Otherwise                                  |

|                                                |                                                                                                                                             |
|------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
|                                                | returns false.                                                                                                                              |
| boolean<br>isLetterOrDigit(char<br><i>ch</i> ) | Returns true if the character passed into <i>ch</i> contains a digit (0 through 9) or an alphabetic letter. Otherwise returns false.        |
| boolean<br>isLowerCase(char <i>ch</i> )        | Returns true if the argument passed into <i>ch</i> is a lowercase letter. Otherwise returns false.                                          |
| boolean<br>isUpperCase(char <i>ch</i> )        | Returns true if the argument passed into <i>ch</i> is an uppercase letter. Otherwise returns false.                                         |
| boolean<br>isSpaceChar(char <i>ch</i> )        | Returns true if the argument passed into <i>ch</i> is a space character. Otherwise returns false.                                           |
| boolean<br>isWhiteSpace(char<br><i>ch</i> )    | Returns true if the argument passed into <i>ch</i> is a whitespace character (a space, tab, or newline character). Otherwise returns false. |

The program in [Code Listing 8-1](#) demonstrates many of these methods.

## Code Listing 8-1 (CharacterTest.java)

```

1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates some of the Character class's
5 * character testing methods.
6 */
7
8 public class CharacterTest
9 {
10 public static void main(String[] args)
11 {
12 String inputLine; // A line of input
13 char inputChar; // A character
14

```

```
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get a character from the user.
19 System.out.print("Enter a character: ");
20 inputLine = keyboard.nextLine();
21 inputChar = inputLine.charAt(0);
22
23 // Test the character.
24 if (Character.isLetter(inputChar))
25 System.out.println("Letter");
26
27 if (Character.isDigit(inputChar))
28 System.out.println("Digit");
29
30 if (Character.isLowerCase(inputChar))
31 System.out.println("Lowercase letter");
32
33 if (Character.isUpperCase(inputChar))
34 System.out.println("Uppercase letter");
35
36 if (Character.isSpaceChar(inputChar))
37 System.out.println("Space");
38
39 if (Character.isWhitespace(inputChar))
40 System.out.println("Whitespace");
41 }
42 }
```

## Program Output with Example Input Shown in Bold

Enter a character: **a**   
Letter  
Lowercase letter

## Program Output with Example Input Shown in Bold

Enter a character: **A**   
Letter  
Uppercase letter

## Program Output with Example Input Shown in Bold

Enter a character: **4**   
Digit

## **Program Output with Example Input Shown in Bold**

Enter a character: **Space**  
Whitespace character

## **Program Output with Example Input Shown in Bold**

Enter any character: **Space**  
Whitespace character

[Code Listing 8-2](#) shows a more practical application of the character testing methods. It tests a string to determine whether it is a seven-character customer number in the proper format.

# **Code Listing 8-2 (CustomerNumber.java)**

```
1 import java.util.Scanner;
2
3 /**
4 * This program tests a customer number to determine
5 * whether it is in the proper format.
6 */
7
8 public class CustomerNumber
9 {
10 public static void main(String[] args)
11 {
12 String customer; // To hold a customer number
13
14 // Create a Scanner object for keyboard input.
15 Scanner keyboard = new Scanner(System.in);
16
17 System.out.println("Enter a customer number in " +
18 "the form LLLNNNN");
19 System.out.print("(LLL = letters and NNNN " +
20 "= numbers): ");
21
22 // Get a customer number from the user.
```

```
23 customer = keyboard.nextLine();
24
25 // Determine whether it is valid.
26 if (isValid(customer))
27 {
28 System.out.println("That's a valid customer " +
29 "number.");
30 }
31 else
32 {
33 System.out.println("That is not the proper " +
34 "format.");
35 System.out.println("Here is an example: " +
36 "ABC1234");
37 }
38 }
39
40 /**
41 * The isValid method accepts a String as its argument
42 * and tests its contents for a valid customer number.
43 */
44
45 private static boolean isValid(String custNumber)
46 {
47 boolean goodSoFar = true; // Flag
48 int index = 0; // Loop control variable
49
50 // Is the string the correct length?
51 if (custNumber.length() != 7)
52 goodSoFar = false;
53
54 // Test the first three characters for letters.
55 while (goodSoFar && index < 3)
56 {
57 if (!Character.isLetter(custNumber.charAt(index)))
58 goodSoFar = false;
59 index++;
60 }
61
62 // Test the last four characters for digits.
63 while (goodSoFar && index < 7)
64 {
65 if (!Character.isDigit(custNumber.charAt(index)))
66 goodSoFar = false;
67 index++;
68 }
69
70 // Return the results
```

```
71 return goodSoFar;
72 }
73 }
```

## Program Output with Example Input Shown in Bold

Enter a customer number in the form LLLNNNN  
(LLL = letters and NNNN = numbers): **RQS4567**   
That's a valid customer number.

## Program Output with Example Input Shown in Bold

Enter a customer number in the form LLLNNNN  
(LLL = letters and NNNN = numbers): **AX467T9**   
That is not the proper format.  
Here is an example: ABC1234

In this program, the customer number is expected to be seven characters in length, and consist of three alphabetic letters followed by four numeric digits. The `isValid` method, in lines 45 through 72, accepts a `String` argument that will be tested. In lines 47 and 48, two local variables are declared: `goodSoFar`, a boolean that is initialized as `true`; and `index`, an `int` that is initialized as 0. The `goodSoFar` variable is a flag that will be set to `false` immediately when the method determines the customer number is not in a valid format. The `index` variable is a loop control variable.

In line 51, the `isValid` method tests the length of the `custNumber` argument. If the argument is not seven characters long, it is not valid, and the `goodSoFar` variable is set to `false` in line 52. Next, the method uses the `while` loop in lines 55 through 60 to validate the first three characters. Recall from [Chapter 2](#) that the `String` class's `charAt` method returns a character at a specific position in a string (position numbering starts at 0). Inside the loop, the `if` statement in line 57 uses the `Character.isLetter` method to test the characters at positions 0, 1, and 2 in the `custNumber` string. If any of these characters are not letters, the `goodSoFar` variable is set to `false` (in line 58) and the loop terminates.

Next, the method uses the `while` loop in lines 63 through 68 to validate the last four characters. Inside the loop, the `if` statement in line 65 uses the

`Character.isDigit` method to test the characters at positions 3, 4, 5, and 6 in the `custNumber` string. If any of these characters are not digits, the `goodSoFar` variable is set to `false` (in line 66) and the loop terminates.

Last, in line 71, the method returns the value of the `goodSoFar` variable.

## Character Case Conversion

The `Character` class also provides the static methods listed in [Table 8-2](#) for converting the case of a character. Each method accepts a `char` argument and returns a `char` value.

### Table 8-2 Some `Character` class methods for case conversion

| Method                                 | Description                                                                  |
|----------------------------------------|------------------------------------------------------------------------------|
| <code>char toLowerCase(char ch)</code> | Returns the lowercase equivalent of the argument passed to <code>ch</code> . |
| <code>char toUpperCase(char ch)</code> | Returns the uppercase equivalent of the argument passed to <code>ch</code> . |

If the `toLowerCase` method's argument is an uppercase character, the method returns the lowercase equivalent. For example, the following statement will display the character `a` on the screen:

```
System.out.println(Character.toLowerCase('A'));
```

If the argument is already lowercase, the `toLowerCase` method returns it unchanged. The following statement also causes the lowercase character `a` to be displayed:

```
System.out.println(Character.toLowerCase('a'));
```

If the `toUpperCase` method's argument is a lowercase character, the method returns the uppercase equivalent. For example, the following statement will display the character A on the screen:

```
System.out.println(Character.toUpperCase('a'));
```

If the argument is already uppercase, the `toUpperCase` method returns it unchanged.

Any nonletter argument passed to `toLowerCase` or `toUpperCase` is returned as it is. Each of the following statements displays the method argument without any change:

```
System.out.println(Character.toLowerCase('*'));
System.out.println(Character.toLowerCase('$'));
System.out.println(Character.toUpperCase('&'));
System.out.println(Character.toUpperCase('%'));
```

The program in [Code Listing 8-3](#) demonstrates the `toUpperCase` method in a loop that asks the user to enter Y or N.

## Code Listing8-3 (circleArea.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the Character
5 * class's toUpperCase method.
6 */
7
8 public class CircleArea
9 {
10 public static void main(String[] args)
11 {
12 double radius; // The circle's radius
13 double area; // The circle's area
14 String input; // To hold a line of input
15 char choice; // To hold a single character
16
17 // Create a Scanner object to read keyboard input.
18 Scanner keyboard = new Scanner(System.in);
19 }
```

```

20 do
21 {
22 // Get the circle's radius.
23 System.out.print("Enter the circle's radius: ");
24 radius = keyboard.nextDouble();
25
26 // Consume the remaining newline character.
27 keyboard.nextLine();
28
29 // Calculate and display the area.
30 area = Math.PI * radius * radius;
31 System.out.printf("The area is %.2f.\n", area);
32
33 // Repeat this?
34 System.out.print("Do you want to do this " +
35 "again? (Y or N) ");
36 input = keyboard.nextLine();
37 choice = input.charAt(0);
38
39 } while (Character.toUpperCase(choice) == 'Y');
40 }
41 }
```

## Program Output with Example Input Shown in Bold

Enter the circle's radius: **10**   
The area is 314.16.

Do you want to do this again? (Y or N) **y** 

Enter the circle's radius: **15**   
The area is 706.86.

Do you want to do this again? (Y or N) **n** 



## Checkpoint

1. 8.1 Write a statement that converts the contents of the char variable `big` to lowercase. The converted value should be assigned to the variable `little`.
2. 8.2 Write an `if` statement that displays the word “digit” if the char variable `ch` contains a numeric digit. Otherwise, it should display “Not a

digit.”

3. 8.3 What is the output of the following statement?

```
System.out.println(Character.toUpperCase(Character.toLowerCase('a')));
```

4. 8.4 Write a loop that asks the user “Do you want to repeat the program or quit? (R/Q)”. The loop should repeat until the user has entered an R or Q (either upper case or lowercase).

5. 8.5 What will the following code display?

```
char var = '$';
System.out.println(Character.toUpperCase(var));
```

6. 8.6 Write a loop that counts the number of uppercase characters that appear in the String object str.

# 8.3 More about String Objects

## Concept:

The `String` class provides several methods for searching and working with `String` objects.

## Searching for Substrings

The `String` class provides several methods that search for a string inside of a string. The term *substring* commonly is used to refer to a string that is part of another string. [Table 8-3](#) summarizes some of these methods. Each of the methods in [Table 8-3](#) returns a boolean value indicating whether the string was found.

### Table 8-3 `String` methods that search for a substring

| Method                                                                            | Description                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean startsWith(String str)</code>                                       | This method returns <code>true</code> if the calling string begins with the string passed into <code>str</code> . Otherwise it returns <code>false</code> .                                                       |
| <code>boolean endsWith(String str)</code>                                         | This method returns <code>true</code> if the calling string ends with the string passed into <code>str</code> . Otherwise it returns <code>false</code> .                                                         |
| <code>boolean regionMatches(int start, String str, int start2, int length)</code> | This method returns <code>true</code> if a specified region of the calling string matches a specified region of the string passed into <code>str</code> . The <code>start</code> parameter indicates the starting |

```
start, String
str, int start2,
int n)
```

position of the region within the calling string. The *start2* parameter indicates the starting position of the region within *str*. The *n* parameter indicates the number of characters in both regions.

```
boolean
regionMatches
(boolean
ignoreCase, int
start, String
str, int start2,
int n)
```

This overloaded version of the *regionMatches* method has an additional parameter, *ignoreCase*. If *true* is passed into this parameter, the method ignores the case of the calling string and *str* when comparing the regions. If *false* is passed into the *ignoreCase* parameter, the comparison is case sensitive.

Let's take a closer look at each of these methods.

## The `startsWith` and `endsWith` Methods

The `startsWith` method determines whether the calling object's string begins with a specified substring. For example, the following code determines whether the string "Four score and seven years ago" begins with "Four". The method returns `true` if the string does begin with the specified substring, or `false` otherwise.

```
String str = "Four score and seven years ago";
if (str.startsWith("Four"))
 System.out.println("The string starts with Four.");
else
 System.out.println("The string does not start with Four.");
```

In the code, the method call `str.startsWith("Four")` returns `true` because the string does begin with "Four". The `startsWith` method performs a case-sensitive comparison, so the method call `str.startsWith("four")` would return `false`.

The `endswith` method determines whether the calling string ends with a specified substring. For example, the following code determines whether the string “Four score and seven years ago” ends with “ago”. The method returns true if the string does end with the specified substring or false otherwise.

```
String str = "Four score and seven years ago";
if (str.endsWith("ago"))
 System.out.println("The string ends with ago.");
else
 System.out.println("The string does not end with ago.");
```

In the code, the method call `str.endsWith("ago")` returns true because the string does end with “ago”. The `endsWith` method also performs a case-sensitive comparison, so the method call `str.endsWith("Ago")` would return false.

The program in [Code Listing 8-4](#) demonstrates a search algorithm that uses the `startsWith` method. The program searches an array of strings for an element that starts with a specified string.

## **Code Listing 8-4 (PersonSearch.java)**

```
18 "Smart, Kathryn", "Smith, Chris",
19 "Smith, Brad", "Williams, Jean" };
20
21 // Create a Scanner object for keyboard input.
22 Scanner keyboard = new Scanner(System.in);
23
24 // Get a partial name to search for.
25 System.out.print("Enter the first few characters of " +
26 "the last name to look up: ");
27 lookUp = keyboard.nextLine();
28
29 // Display all of the names that begin with the
30 // string entered by the user.
31 System.out.println("Here are the names that match:");
32 for (String person : people)
33 {
34 if (person.startsWith(lookUp))
35 System.out.println(person);
36 }
37 }
38 }
```

## Program Output with Example Input Shown in Bold

```
Enter the first few characters of the last name to look up: Davis
Here are the names that match:
Davis, George
Davis, Jenny
```

## Program Output with Example Input Shown in Bold

```
Enter the first few characters of the last name to look up: Russ
Here are the names that match:
Russert, Phil
Russell, Cindy
```

# The regionMatches Methods

The `String` class provides overloaded versions of the `regionMatches` method, which determines whether specified regions of two strings match. The following code demonstrates this method:

```

String str = "Four score and seven years ago";
String str2 = "Those seven years passed quickly";
if (str.regionMatches(15, str2, 6, 11))
 System.out.println("The regions match.");
else
 System.out.println("The regions do not match.");

```

This code will display “The regions match.” The specified region of the str string begins at position 15, and the specified region of the str2 string begins at position 6. Both regions consist of 11 characters. The specified region in the str string is “seven years” and the specified region in the str2 string is also “seven years”. Because the two regions match, the regionMatches method in this code returns true. This version of the regionMatches method performs a case-sensitive comparison. An overloaded version accepts an additional argument indicating whether to perform a case-insensitive comparison. The following code demonstrates this method:

```

String str = "Four score and seven years ago";
String str2 = "THOSE SEVEN YEARS PASSED QUICKLY";

if (str.regionMatches(true, 15, str2, 6, 11))
 System.out.println("The regions match.");
else
 System.out.println("The regions do not match.");

```

This code will also display “The regions match.” The first argument passed to this version of the regionMatches method can be true or false, indicating whether a case-insensitive comparison should be performed. In this example, true is passed, so case will be ignored when the regions “seven years” and “SEVEN YEARS” are compared.

Each of these methods indicates by a boolean return value whether a substring appears within a string. The String class also provides methods that not only search for items within a string, but report the location of those items. [Table 8-4](#) describes overloaded versions of the indexOf and lastIndexOf methods.

## Table 8-4 String methods for

# getting a character or substring's location

| Method                                           | Description                                                                                                                                                                                                                                                                                                  |
|--------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int indexOf(char ch)</code>                | Sets the calling <code>String</code> object for the character passed into <code>ch</code> . If the character is found, the position of its first occurrence is returned. Otherwise, -1 is returned.                                                                                                          |
| <code>int indexOf(char ch, int start)</code>     | Sets the calling <code>String</code> object for the character passed into <code>ch</code> , beginning at the position passed into <code>start</code> and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, -1 is returned.             |
| <code>int indexOf(String str)</code>             | Sets the calling <code>String</code> object for the string passed into <code>str</code> . If the string is found, the beginning position of its first occurrence is returned. Otherwise, -1 is returned.                                                                                                     |
| <code>int indexOf(String str, int start)</code>  | Sets the calling <code>String</code> object for the string passed into <code>str</code> . The search begins at the position passed into <code>start</code> and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, -1 is returned. |
| <code>int lastIndexOf(char ch)</code>            | Sets the calling <code>String</code> object for the character passed into <code>ch</code> . If the character is found, the position of its last occurrence is returned. Otherwise, -1 is returned.                                                                                                           |
| <code>int lastIndexOf(char ch, int start)</code> | Sets the calling <code>String</code> object for the character passed into <code>ch</code> , beginning at the position passed into <code>start</code> . The search is conducted backward through the string, to position 0. If the character is found, the                                                    |

|                                              |                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                              | position of its last occurrence is returned.<br>Otherwise, -1 is returned.                                                                                                                                                                                                                                                                 |
| int<br>lastIndexOf(String<br>str)            | Searches the calling <code>String</code> object for the string passed into <code>str</code> . If the string is found, the beginning position of its last occurrence is returned. Otherwise, -1 is returned.                                                                                                                                |
| int<br>lastIndexOf(String<br>str, int start) | Searches the calling <code>String</code> object for the string passed into <code>str</code> , beginning at the position passed into <code>start</code> . The search is conducted backward through the string, to position 0. If the string is found, the beginning position of its last occurrence is returned. Otherwise, -1 is returned. |

## Finding Characters with the `indexOf` and `lastIndexOf` Methods

The `indexOf` and `lastIndexOf` methods can search for either a character or a substring within the calling string. If the item being searched for is found, its position is returned. Otherwise, -1 is returned. Here is an example of code using two of the methods to search for a character:

```
String str = "Four score and seven years ago";
int first, last;

first = str.indexOf('r');
last = str.lastIndexOf('r');

System.out.println("The letter r first appears at " +
 "position " + first);

System.out.println("The letter r last appears at " +
 "position " + last);
```

This code produces the following output:

```
The letter r first appears at position 3
```

The letter r last appears at position 24

The following code shows another example. It uses a loop to show the positions of each letter “r” in the string.

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
 "following locations:");
position = str.indexOf('r');
while (position != -1)
{
 System.out.println(position);
 position = str.indexOf('r', position + 1);
}
```

This code will produce the following output:

```
The letter r appears at the following locations:
3
8
24
```

The following code is very similar, but it uses the `lastIndexOf` method and shows the positions in reverse order.

```
String str = "Four score and seven years ago";
int position;

System.out.println("The letter r appears at the " +
 "following locations.");

position = str.lastIndexOf('r');
while (position != -1)
{
 System.out.println(position);
 position = str.lastIndexOf('r', position - 1);
}
```

This code will produce the following output:

The letter r appears at the following locations.  
24  
8  
3

## Finding Substrings with the `indexOf` and `lastIndexOf` Methods

The `indexOf` and `lastIndexOf` methods can also search for substrings within a string. The following code shows an example. It displays the starting positions of each occurrence of the word “and” within a string.

```
String str = "and a one and a two and a three";
int position;

System.out.println("The word and appears at the " +
 "following locations.");
position = str.indexOf("and");
while (position != -1)
{
 System.out.println(position);
 position = str.indexOf("and", position + 1);
}
```

This code produces the following output:

```
The word and appears at the following locations.
0
10
20
```

The following code also displays the same results, but in reverse order:

```
String str = "and a one and a two and a three";
int position;

System.out.println("The word and appears at the " +
 "following locations.");
position = str.lastIndexOf("and");
while (position != -1)
{
```

```
 System.out.println(position);
 position = str.lastIndexOf("and", position - 1);
 }
```

This code produces the following output:

```
The word and appears at the following locations.
20
10
0
```

## Extracting Substrings

The `String` class provides several methods that allow you to retrieve a substring from a string. The methods we will examine are listed in [Table 8-5](#).

## Table 8-5 String methods for extracting substrings

| Method                                            | Description                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String substring(int start)</code>          | This method returns a copy of the substring that begins at <i>start</i> and goes to the end of the calling object's string.                                                                                                                                                                                                               |
| <code>String substring(int start, int end)</code> | This method returns a copy of a substring. The argument passed into <i>start</i> is the substring's starting position, and the argument passed into <i>end</i> is the substring's ending position. The character at the <i>start</i> position is included in the substring, but the character at the <i>end</i> position is not included. |
|                                                   | This method extracts a substring from the calling object and stores it in a <code>char</code> array. The argument passed into <i>start</i> is the substring's starting position, and the argument passed into <i>end</i> is the                                                                                                           |

```
void
getChars(int
start, int
end, char[]
array, int
arrayStart)
```

substring's ending position. The character at the start position is included in the substring, but the character at the end position is not included. (The last character in the substring ends at `end - 1`.) The characters in the substring are stored as elements in the array that is passed into the `array` parameter. The `arrayStart` parameter specifies the starting subscript within the array where the characters are to be stored.

`char[] toCharArray()` This method returns all of the characters in the calling object as a `char` array.

## The `substring` Methods

The `substring` method returns a copy of a substring from the calling object. There are two overloaded versions of this method. The first version accepts an `int` argument that is the starting position of the substring. The method returns a string consisting of all the characters from the starting position to the end of the string. The character at the starting position is part of the substring. Here is an example of the method's use.

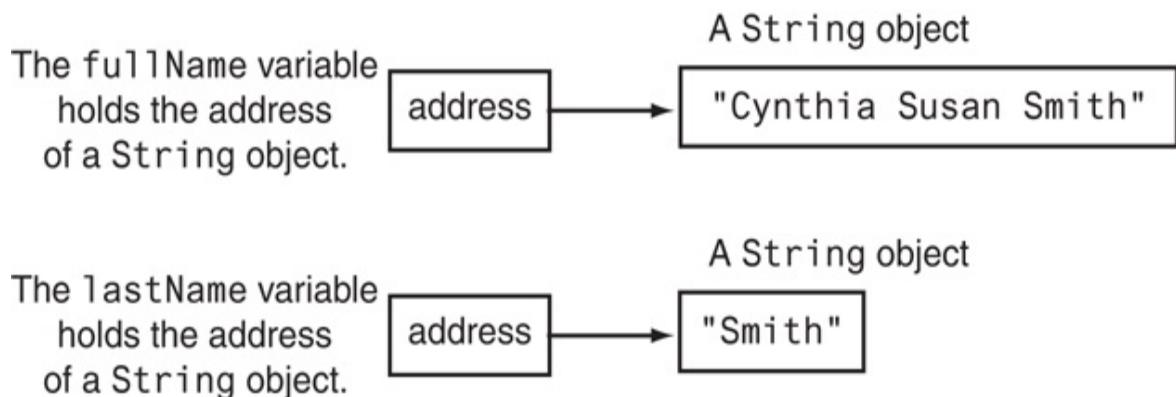
```
String fullName = "Cynthia Susan Smith";
String lastName = fullName.substring(14);
System.out.println("The full name is " + fullName);
System.out.println("The last name is " + lastName);
```

This code will produce the following output:

```
The full name is Cynthia Susan Smith
The last name is Smith
```

Keep in mind that the `substring` method returns a new `String` object that holds a copy of the substring. When this code executes, the `fullName` and `lastName` variables will reference two different `String` objects as shown in [Figure 8-1](#).

# Figure 8-1 The `fullName` and `lastName` variables reference separate objects



[Figure 8-1 Full Alternative Text](#)

The second version of the method accepts two `int` arguments. The first specifies the substring's starting position, and the second specifies the substring's ending position. The character at the starting position is included in the substring, but the character at the ending position is not. Here is an example of how the method is used:

```
String fullName = "Cynthia Susan Smith";
String middleName = fullName.substring(8, 13);
System.out.println("The full name is " + fullName);
System.out.println("The middle name is " + middleName);
```

The code will produce the following output:

```
The full name is Cynthia Susan Smith
The middle name is Susan
```

## The `getChars` and `toCharArray` Methods

The `getChars` and `toCharArray` methods both convert the calling `String` object to a `char` array. The `getChars` method can be used to convert a substring, whereas the `toCharArray` method converts the entire string. Here is an example of how the `getChars` method might be used:

```
String fullName = "Cynthia Susan Smith";
char[] nameArray = new char[5];
fullName.getChars(8, 13, nameArray, 0);
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
 System.out.print(nameArray[i] + " ");
```

This code stores the individual characters of the substring “Susan” in the elements of the `nameArray` array, beginning at element 0. The code will produce the following output:

```
The full name is Cynthia Susan Smith
The values in the array are:
S u s a n
```

The `toCharArray` method returns a reference to a `char` array that contains all of the characters in the calling object. Here is an example:

```
String fullName = "Cynthia Susan Smith";
char[] nameArray;
nameArray = fullName.toCharArray();
System.out.println("The full name is " + fullName);
System.out.println("The values in the array are:");
for (int i = 0; i < nameArray.length; i++)
 System.out.print(nameArray[i] + " ");
```

This code will produce the following output:

```
The full name is Cynthia Susan Smith
The values in the array are:
C y n t h i a S u s a n S m i t h
```

These methods can be used when you want to use an array processing algorithm on the contents of a `String` object. The program in [Code Listing 8-5](#) converts a `String` object to an array, then uses the array to determine the number of letters, digits, and whitespace characters in the string.

# Code Listing 8-5

## (StringAnalyzer.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program displays the number of letters, digits, and
5 * whitespace characters in a string.
6 */
7
8 public class StringAnalyzer
9 {
10 public static void main(String[] args)
11 {
12 String str; // To hold the input as a string
13 char[] array; // To hold the input as an array
14 int letters = 0, // Total number of alphabetic letters
15 digits = 0, // Total number of digits
16 whitespaces = 0; // Total number of whitespace character
17
18 // Create a Scanner object for keyboard input.
19 Scanner keyboard = new Scanner(System.in);
20
21 // Get a string from the user.
22 System.out.print("Enter a string: ");
23 str = keyboard.nextLine();
24
25 // Convert the string to a char array.
26 array = str.toCharArray();
27
28 // Analyze the characters.
29 for (int i = 0; i < array.length; i++)
30 {
31 if (Character.isLetter(array[i]))
32 letters++;
33 else if (Character.isDigit(array[i]))
34 digits++;
35 else if (Character.isWhitespace(array[i]))
36 whitespaces++;
37 }
38
39 // Display the results.
40 System.out.println("That string contains " +
41 letters + " letters, " +
```

```

42 digits + " digits, and " +
43 whitespaces +
44 " whitespace characters.");
45 }
46 }

```

## Program Output with Example Input Shown in Bold

Enter a string: **99 red balloons**  That string contains 11 letters, 2 digits, and 2 whitespace chara

# Methods That Return a Modified String

The String class methods listed in [Table 8-6](#) return a modified copy of a String object.

## Table 8-6 Methods that return a modified copy of a string object

| Method                                                  | Description                                                                                                                                                                                    |
|---------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String concat(String str)</code>                  | This method returns a copy of the calling String object with the contents of <i>str</i> concatenated to it.                                                                                    |
| <code>String replace(char oldChar, char newChar)</code> | This method returns a copy of the calling String object, in which all occurrences of the character passed into <i>oldChar</i> have been replaced by the character passed into <i>newChar</i> . |
| <code>String trim()</code>                              | This method returns a copy of the calling String object, in which all leading and trailing whitespace characters have been deleted.                                                            |

The concat method performs the same operation as the + operator when used with strings. For example, look at the following code, which uses the + operator:

```
String fullName,
 firstName = "Timothy ",
 lastName = "Haynes";
fullName = firstName + lastName;
```

Equivalent code can also be written with the concat method. Here is an example:

```
String fullName,
 firstName = "Timothy ",
 lastName = "Haynes";
fullName = firstName.concat(lastName);
```

The replace method returns a copy of a String object, where every occurrence of a specified character has been replaced with another character. For example, look at the following code:

```
String str1 = "Tom Talbert Tried Trains";
String str2;
str2 = str1.replace('T', 'D');
System.out.println(str1);
System.out.println(str2);
```

In this code, the replace method will return a copy of the str1 object with every occurrence of the letter “T” replaced with the letter “D”. The code will produce the following output:

```
Tom Talbert Tried Trains
Dom Dalbert Dried Drains
```

Remember that the replace method does not modify the contents of the calling String object, but returns a modified copy of it. After the previous code executes, the str1 and str2 variables will reference different String objects.

The trim method returns a copy of a String object with all leading and trailing whitespace characters deleted. A *leading* whitespace character is one that appears at the beginning, or left side, of a string. For example, the

following string has three leading whitespace characters:

```
" Hello"
```

A *trailing* whitespace character is one that appears at the end, or right side, of a string, after the nonspace characters. For example, the following string has three trailing whitespace characters:

```
"Hello "
```

Here is an example:

```
String greeting1 = " Hello ";
String greeting2;
greeting2 = greeting1.trim();
System.out.println("*" + greeting1 + "*");
System.out.println("*" + greeting2 + "*");
```

In this code, the first statement assigns the string " Hello " (with three leading spaces and three trailing spaces) to the greeting1 variable. The trim method is called, which returns a copy of the string with the leading and trailing spaces removed. The code will produce the following output:

```
* Hello *
Hello
```

One common use of the trim method is to remove any leading or trailing spaces the user might have entered while inputting data.

## The Static valueOf Methods

The String class has several overloaded versions of a method named valueOf. This method accepts a value of any primitive data type as its argument and returns a string representation of the value. [Table 8-7](#) describes these methods.

## Table 8-7 Some of the String

# class's valueOf methods

| Method                                                                         | Description                                                                                                                                                                                                                                            |
|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| String valueOf(boolean <i>b</i> )                                              | If the boolean argument passed to <i>b</i> is true, the valueOf method returns the string “true”. If the argument is false, the method returns the string “false”.                                                                                     |
| String valueOf(char <i>c</i> )                                                 | This method returns a string containing the character passed into <i>c</i> .                                                                                                                                                                           |
| String valueOf(char[] <i>array</i> )                                           | This method returns a string that contains all of the elements in the char array passed into <i>array</i> .                                                                                                                                            |
| String valueOf(char[] <i>array</i> , int <i>subscript</i> , int <i>count</i> ) | This method returns a string that contains part of the elements in the char array passed into <i>array</i> . The argument passed into <i>subscript</i> is the starting subscript, and the argument passed into <i>count</i> is the number of elements. |
| String valueOf(double <i>number</i> )                                          | This method returns the string representation of the double argument passed into <i>number</i> .                                                                                                                                                       |
| String valueOf(float <i>number</i> )                                           | This method returns the string representation of the float argument passed into <i>number</i> .                                                                                                                                                        |
| String valueOf(int <i>number</i> )                                             | This method returns the string representation of the int argument passed into <i>number</i> .                                                                                                                                                          |
| String valueOf(long <i>number</i> )                                            | This method returns the string representation of the long argument passed into <i>number</i> .                                                                                                                                                         |

The following code demonstrates several of these methods.

```
boolean b = true;
char[] letters = { 'a', 'b', 'c', 'd', 'e' };
double d = 2.4981567;
int i = 7;

System.out.println(String.valueOf(b));
System.out.println(String.valueOf(letters));
System.out.println(String.valueOf(letters, 1, 3));
System.out.println(String.valueOf(d));
```

```
System.out.println(String.valueOf(i));
```

This code will produce the following output:

```
true
abcde
bcd
2.4981567
7
```



## Checkpoint

1. 8.7 Write a method that accepts a `String` object as an argument and returns `true` if the argument ends with the substring “ger”. Otherwise, the method should return `false`.
2. 8.8 Modify the method you wrote for [Checkpoint 8.7](#) so it performs a case-insensitive test. The method should return `true` if the argument ends with “ger” in any possible combination of upper and lowercase letters.
3. 8.9 Look at the following declaration:

```
String cafeName = "Broadway Cafe";
String str;
```

Which of the following methods would you use to make `str` reference the string "Broadway"?

1. `startsWith`
  2. `regionMatches`
  3. `substring`
  4. `indexOf`
4. 8.10 What is the difference between the `indexOf` and `lastIndexOf`

methods?

5. 8.11 What is the difference between the `getChars` and `substring` methods?
6. 8.12 The `+` operator, when used with strings, performs the same operation as what `String` method?
7. 8.13 What is the difference between the `getChars` and `toCharArray` methods?
8. 8.14 Look at the following code:

```
String str1 = "To be, or not to be";
String str2 = str1.replace('o', 'u');
System.out.println(str1);
System.out.println(str2);
```

You hear a fellow student claim that the code will display the following:

Tu be ur nut tu be  
Tu be ur nut tu be

Is your fellow student right or wrong? Why?

9. 8.15 What will the following code display?

```
String str1 = "William ",
 str2 = " the ",
 str3 = " Conqueror";
System.out.println(str1.trim() + str2.trim() +
 str3.trim());
```

10. 8.16 Assume that a program has the following declarations:

```
double number = 9.47;
String str;
```

Write a statement that assigns a string representation of the number variable to `str`.

## 8.4 The StringBuilder Class

### Concept:

The `StringBuilder` class is similar to the `String` class, except that you can change the contents of `StringBuilder` objects. The `StringBuilder` class also provides several useful methods that the `String` class does not have.

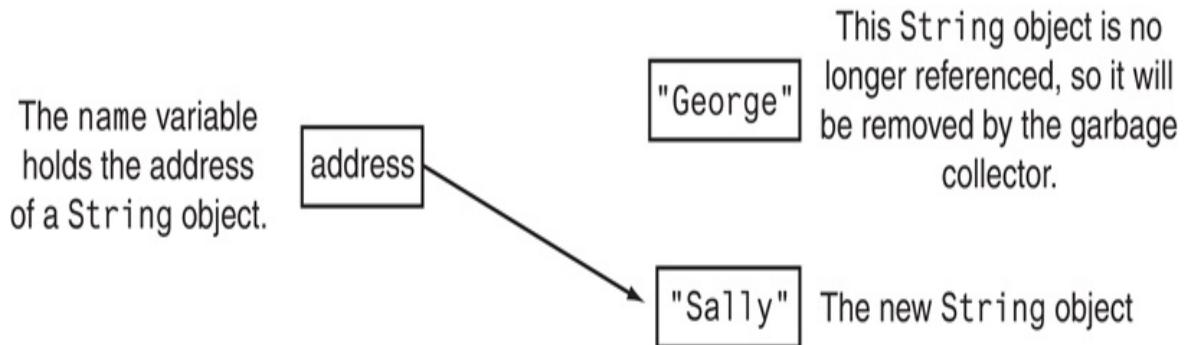
The `StringBuilder` class is similar to the `String` class. The main difference between the two is that you can change the contents of a `StringBuilder` object, but you cannot change the contents of a `String` object. Recall from [Chapter 6](#) that `String` objects are immutable. This means that once you set the contents of a `String` object, you cannot change the string value that it holds. For example, look at the following code:

```
String name;
name = "George";
name = "Sally";
```

The first statement creates the `name` variable. The second creates a `String` object containing the string “George” and assigns its address to the `name` variable. Although we cannot change the contents of the `String` object, we can make the `name` variable reference a different `String` object. That’s what the third statement does: It creates another `String` object containing the string “Sally”, and assigns its address to `name`. This is illustrated by [Figure 8-2](#).

**Figure 8-2 The `String` object containing “George” is no**

# longer referenced



[Figure 8-2 Full Alternative Text](#)

Unlike `String` objects, `StringBuilder` objects have methods that allow you to modify their contents without creating a new object in memory. You can change specific characters, insert characters, delete characters, and perform other operations. The `StringBuilder` object will grow or shrink in size, as needed, to accommodate the changes.

The fact that `String` objects are immutable is rarely a problem, but you might consider using `StringBuilder` objects if your program needs to make a lot of changes to one or more strings. This will improve the program's efficiency by reducing the number of `String` objects that must be created and then removed by the garbage collector. Now, let's look at the `StringBuilder` class's constructors and methods.

## The `StringBuilder` Constructors

[Table 8-8](#) lists three of the `StringBuilder` constructors.

### Table 8-8 `StringBuilder` constructors

| <b>Constructor</b>                     | <b>Description</b>                                                                                                                                   |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>StringBuilder()</code>           | This constructor accepts no arguments. It gives the object enough storage space to hold 16 characters, but no characters are stored in it.           |
| <code>StringBuilder(int length)</code> | This constructor gives the object enough storage space to hold <i>length</i> characters, but no characters are stored in it.                         |
| <code>StringBuilder(String str)</code> | This constructor initializes the object with the string in <i>str</i> . The object's initial storage space will be the length of the string plus 16. |

The first two constructors create empty `StringBuilder` objects of a specified size. The first constructor makes the `StringBuilder` object large enough to hold 16 characters, and the second constructor makes the object large enough to hold *length* characters. Remember, `StringBuilder` objects automatically resize themselves, so it is not a problem if you later want to store a larger string in the object. The third constructor accepts a `String` object as its argument and assigns the object's contents to the `StringBuilder` object. Here is an example of its use:

```
StringBuilder city = new StringBuilder("Charleston");
System.out.println(city);
```

This code creates a `StringBuilder` object and assigns its address to the `city` variable. The object is initialized with the string “Charleston”. As demonstrated by this code, you can pass a `StringBuilder` object to the `println` and `print` methods.

One limitation of the `StringBuilder` class is that you cannot use the assignment operator to assign strings to `StringBuilder` objects. For example, the following code will not work:

```
StringBuilder city = "Charleston"; // ERROR!!! Will not work!
```

Instead of using the assignment operator, you must use the `new` key word and a constructor, or one of the `StringBuilder` methods, to store a string in a

`StringBuilder` object.

## Other `StringBuilder` Methods

The `StringBuilder` class provides many of the same methods as the `String` class. [Table 8-9](#) lists several of the `StringBuilder` methods that work exactly like their `String` class counterparts.

### Table 8-9 Methods that are common to the `String` and `StringBuilder` classes

```
char charAt(int position)
void getChars(int start, int end, char[] array, int arrayStart)
int indexOf(String str)
int indexOf(String str, int start)
int lastIndexOf(String str)
int lastIndexOf(String str, int start)
int length()
String substring(int start)
String substring(int start, int end)
```

In addition, the `StringBuilder` class provides several methods that the `String` class does not have. Let's look at a few of them.

# The append Methods

The `StringBuilder` class has several overloaded versions of a method named `append`. These methods accept an argument that may be of any primitive data type, a char array, or a `String` object. They append a string representation of their argument to the calling object's current contents. Because there are so many overloaded versions of `append`, we will examine the general form of a typical call to the method:

```
object.append(item);
```

After the method is called, a string representation of `item` will be appended to `object`'s contents. The following code shows some of the `append` methods being used:

```
StringBuilder str = new StringBuilder();

// Append values to the object.
str.append("We sold "); // Append a String object.
str.append(12); // Append an int.
str.append(" doughnuts for $"); // Append another String.
str.append(15.95); // Append a double.

// Display the object's contents.
System.out.println(str);
```

This code will produce the following output:

```
We sold 12 doughnuts for $15.95
```

For more variations of the `append` method, see the Java API documentation.

# The insert Methods

The `StringBuilder` class also has several overloaded versions of a method named `insert`, which inserts a value into the calling object's string. These methods accept two arguments: an `int` that specifies the position in the calling object's string where the insertion should begin, and the value to be

inserted. The value to be inserted can be of any primitive data type, a char array, or a String object. Because there are so many overloaded versions of `insert`, we will examine the general form of a typical call to the method.

```
object.insert(start, item);
```

In the general form, `start` is the starting position of the insertion and `item` is the item to be inserted. The following code shows an example:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

The first statement creates a `StringBuilder` object initialized with the string "New City". The second statement inserts the string "York " into the `StringBuilder` object, beginning at position 4. The characters that are currently in the object beginning at position 4 are moved to the right. In memory, the `StringBuilder` object is automatically expanded in size to accommodate the inserted characters. If these statements were in a complete program and we ran it, we would see New York City displayed on the screen.

The following code shows how a char array can be inserted into a `StringBuilder` object:

```
char cArray[] = { '2', '0', ' ' };
StringBuilder str = new StringBuilder("In July we sold cars.");
str.insert(16, cArray);
System.out.println(str);
```

The first statement declares a char array named `cArray` containing the characters '2', '0', and ' '. The second statement creates a `StringBuilder` object initialized with the string "In July we sold cars." The third statement inserts the characters in `cArray` into the `StringBuilder` object, beginning at position 16. The characters that are currently in the object beginning at position 16 are moved to the right. If these statements were in a complete program and we ran it, we would see "In July we sold 20 cars." displayed on the screen.

# The replace Method

The `StringBuilder` class has a `replace` method that differs slightly from the `String` class's `replace` method. Whereas the `String` class's `replace` method replaces the occurrences of one character with another character, the `StringBuilder` class's `replace` method replaces a specified substring with a string. Here is the general form of a call to this method:

```
object.replace(start, end, str);
```

In the general form, `start` is an `int` that specifies the starting position of a substring in the calling object, and `end` is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The `str` parameter is a `String` object. After the method executes, the substring will be replaced with `str`. Here is an example:

```
StringBuilder str =
 new StringBuilder("We moved from Chicago to Atlanta.");
str.replace(14, 21, "New York");
System.out.println(str);
```

The `replace` method in this code replaces the word “Chicago” with “New York”. The code will produce the following output:

```
We moved from New York to Atlanta.
```

# The delete, deleteCharAt, and setCharAt Methods

The `delete` and `deleteCharAt` methods are used to delete a substring or a character from a `StringBuilder` object. The `setCharAt` method changes a specified character to another value. [Table 8-10](#) describes these methods.

# Table 8-10 The `StringBuilder` class's `delete` and `deleteCharAt` methods

| Method                                                 | Description                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>StringBuilder delete(int start, int end)</code>  | The <code>start</code> parameter is an <code>int</code> that specifies the starting position of a substring in the calling object, and the <code>end</code> parameter is an <code>int</code> that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.) The method will delete the substring. |
| <code>StringBuilder deleteCharAt (int position)</code> | The <code>position</code> parameter specifies the location of a character that will be deleted.                                                                                                                                                                                                                                                                               |
| <code>void setCharAt(int position, char ch)</code>     | This method changes the character at <code>position</code> to the value passed into <code>ch</code> .                                                                                                                                                                                                                                                                         |

The following code demonstrates both of these methods.

```
StringBuilder str =
 new StringBuilder("I ate 100 blueberries!");

// Display the StringBuilder object.
System.out.println(str);

// Delete the '0'.
str.deleteCharAt(8);

// Delete "blue".
str.delete(9, 13);

// Display the StringBuilder object.
System.out.println(str);
```

```
// Change the '1' to '5'
str.setCharAt(6, '5');

// Display the StringBuilder object.
System.out.println(str);
```

This code will produce the following output.

```
I ate 100 blueberries!
I ate 10 berries!
I ate 50 berries!
```

Although the `StringBuilder` methods presented in this section generally are the most useful, there are others that we haven't covered. Refer to the Java API documentation for more details.

## The `toString` Method

If you need to convert a `StringBuilder` object to a regular string, you can call the object's `toString` method. The following code shows an example of a `StringBuilder` object's contents being assigned to a `String` variable:

```
StringBuilder strb = new StringBuilder("This is a test.");
String str = strb.toString();
```

## In the Spotlight: Formatting and Unformatting Telephone Numbers



Telephone numbers in the United States are commonly formatted to appear in the following manner:

(XXX)XXX-XXXX

In this format, X represents a digit. The three digits that appear inside the

parentheses are the area code. The three digits following the area code are the prefix, and the four digits after the hyphen are the line number. Here is an example:

(919)555-1212

Although the parentheses and the hyphen make the number easier for people to read, those characters are unnecessary for processing by a computer. In a computer system, a telephone number is commonly stored as an unformatted series of digits, as shown here:

9195551212

A program that works with telephone numbers usually needs to unformat numbers that have been entered by the user. This means that the parentheses and the hyphen must be removed prior to storing the number in a file or processing it in some other way. In addition, such programs need the ability to format a number so it contains the parentheses and the hyphen before displaying it on the screen or printing it on paper.

[Code Listing 8-6](#) shows a class named Telephone that contains the following static methods:

- `isFormatted`. This method accepts a `String` argument and returns `true` if the argument is formatted as `(xxx)xxx-xxxx`. If the argument is not formatted this way, the method returns `false`.
- `unformat`. This method accepts a `String` argument. If the argument is formatted as `(xxx)xxx-xxxx`, the method returns an unformatted version of the argument with the parentheses and the hyphen removed. Otherwise, the method returns the original argument.
- `format`. This method's purpose is to format sequence of digits as `(xxx)xxx-xxxx`. The sequence of digits is passed as a `String` argument. If the argument is 10 characters in length, then the method returns the argument with parentheses and a hyphen inserted. Otherwise, the method returns the original argument.

The program in [Code Listing 8-7](#) demonstrates the Telephone class.

# Code Listing8-6 (Telephone.java)

```
1 /**
2 * The Telephone class provides static methods
3 * for formatting and unformatting US telephone
4 * numbers.
5 */
6
7 public class Telephone
8 {
9 // These constant fields hold the valid lengths of
10 // strings that are formatted and unformatted.
11 public final static int FORMATTED_LENGTH = 13;
12 public final static int UNFORMATTED_LENGTH = 10;
13
14 /**
15 * The isFormatted method accepts a string argument
16 * and determines whether it is properly formatted as
17 * a US telephone number in the following manner:
18 * (XXX)XXX-XXXX
19 * If the argument is properly formatted, the method
20 * returns true, otherwise false.
21 */
22
23 public static boolean isFormatted(String str)
24 {
25 boolean valid; // Flag to indicate valid format
26
27 // Determine whether str is properly formatted.
28 if (str.length() == FORMATTED_LENGTH &&
29 str.charAt(0) == '(' &&
30 str.charAt(4) == ')' &&
31 str.charAt(8) == '-')
32 valid = true;
33 else
34 valid = false;
35
36 // Return the value of the valid flag.
37 return valid;
38 }
39
40 /**
41 * The unformat method accepts a string containing
42 * a telephone number formatted as:
43 * (XXX)XXX-XXXX.
```

```
44 * If the argument is formatted in this way, the
45 * method returns an unformatted string where the
46 * parentheses and hyphen have been removed. Otherwise,
47 * it returns the original argument.
48 */
49
50 public static String unformat(String str)
51 {
52 // Create a StringBuilder initialized with str.
53 StringBuilder strb = new StringBuilder(str);
54
55 // If the argument is properly formatted, then
56 // unformat it.
57 if (isFormatted(str))
58 {
59 // First, delete the left paren at position 0.
60 strb.deleteCharAt(0);
61
62 // Next, delete the right paren. Because of the
63 // previous deletion it is now located at
64 // position 3.
65 strb.deleteCharAt(3);
66
67 // Next, delete the hyphen. Because of the
68 // previous deletions it is now located at
69 // position 6.
70 strb.deleteCharAt(6);
71 }
72
73 // Return the unformatted string.
74 return strb.toString();
75 }
76
77 /**
78 * The format method formats a string as:
79 * (XXX)XXX-XXXX.
80 * If the length of the argument is UNFORMATTED_LENGTH
81 * the method returns the formatted string. Otherwise,
82 * it returns the original argument.
83 */
84
85 public static String format(String str)
86 {
87 // Create a StringBuilder initialized with str.
88 StringBuilder strb = new StringBuilder(str);
89
90 // If the argument is the correct length, then
91 // format it.
```

```
92 if (str.length() == UNFORMATTED_LENGTH)
93 {
94 // First, insert the left paren at position 0.
95 strb.insert(0, "(");
96
97 // Next, insert the right paren at position 4.
98 strb.insert(4, ")");
99
100 // Next, insert the hyphen at position 8.
101 strb.insert(8, "-");
102 }
103
104 // Return the formatted string.
105 return strb.toString();
106}
107}
```

## Code Listing8-7 (TelephoneTester.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the Telephone
5 * class's static methods.
6 */
7
8 public class TelephoneTester
9 {
10 public static void main(String[] args)
11 {
12 String phoneNumber; // To hold a phone number
13
14 // Create a Scanner object for keyboard input.
15 Scanner keyboard = new Scanner(System.in);
16
17 // Get an unformatted telephone number.
18 System.out.print("Enter an unformatted telephone number: ")
19 phoneNumber = keyboard.nextLine();
20
21 // Format the telephone number.
22 System.out.println("Formatted: " +
23 Telephone.format(phoneNumber));
24
```

```
25 // Get a formatted telephone number.
26 System.out.println("Enter a telephone number formatted as")
27 System.out.print("(XXX)XXX-XXXX : ");
28 phoneNumber = keyboard.nextLine();
29
30 // Unformat the telephone number.
31 System.out.println("Unformatted: " +
32 Telephone.unformat(phoneNumber));
33 }
34 }
```

## Program Output with Example Input Shown in Bold

```
Enter an unformatted telephone number: 9195551212 Enter
Formatted: (919)555-1212
Enter a telephone number formatted as
(XXX)XXX-XXXX: (828)555-1212 Enter
Unformatted: 8285551212
```



## Note:

The Java API provides a class named `StringBuffer` that is essentially the same as the `StringBuilder` class, with the same constructors and the same methods. The difference is that the methods in the `StringBuffer` class are *synchronized*. This means that the `StringBuffer` class is safe to use in a multithreaded application. Multithreaded programming is beyond the scope of this book, but in a nutshell, a *multithreaded application* is one that concurrently runs multiple threads of execution. In such an application, more than one thread can access the same objects in memory at the same time. In multithreaded applications, the methods must be synchronized to prevent the possibility of data corruption.

Because synchronization requires extra steps to be performed, the `StringBuffer` class is slower than the `StringBuilder` class. In an application where the object will not be accessed by multiple threads, you should use the `StringBuilder` class to get the best performance. In an application where multiple threads will be accessing the object, you should use the `StringBuffer` class to ensure that its data does not become corrupted.



# Checkpoint

1. 8.17 The `String` class is immutable. What does this mean?
2. 8.18 In a program that makes lots of changes to strings, would it be more efficient to use `String` objects or `StringBuilder` objects? Why?
3. 8.19 Look at the following statement:

```
String city = "Asheville";
```

Rewrite this statement so that `city` is a `StringBuilder` object instead of a `String` object.

4. 8.20 You wish to add a string to the end of the existing contents of a `StringBuilder` object. What method do you use?
5. 8.21 You wish to insert a string into the existing contents of a `StringBuilder` object. What method do you use?
6. 8.22 You wish to delete a specific character from the existing contents of a `StringBuilder` object. What method do you use?
7. 8.23 You wish to change a specific character in a `StringBuilder` object. What method do you use?
8. 8.24 How does the `StringBuilder` class's `replace` method differ from the `String` class's `replace` method?

# 8.5 Tokenizing Strings

## Concept:

Tokenizing a string is a process of breaking a string down into its components, which are called tokens. The StringTokenizer class and the String class's split method can be used to tokenize strings.

Sometimes a string will contain a series of words or other items of data separated by spaces or other characters. For example, look at the following string:

"peach raspberry strawberry vanilla"

This string contains the following four items of data: peach, raspberry, strawberry, and vanilla. In programming terms, items such as these are known as *tokens*. Notice that a space appears between the items. The character that separates tokens is known as a *delimiter*. Here is another example:

"17;92;81;12;46;5"

This string contains the following tokens: 17, 92, 81, 12, 46, and 5. Notice that a semicolon appears between each item. The semicolon is used as a delimiter. Some programming problems require you to read a string that contains a list of items then extract all of the tokens from the string for processing. For example, look at the following string that contains a date:

"11-22-2018"

The tokens in this string are 11, 22, and 2018, and the delimiter is the hyphen character. Perhaps a program needs to extract the month, day, and year from such a string. Another example is an operating system pathname, such as the

following:

```
/home/rsullivan/data
```

The tokens in this string are home, rsullivan, and data, and the delimiter is the / character. Perhaps a program needs to extract all of the directory names from such a pathname.

The process of breaking a string into tokens is known as *tokenizing*. In this section, we will discuss two of Java's tools for tokenizing strings: the StringTokenizer class and the String class's split method.

## The StringTokenizer Class

The Java API provides a class, StringTokenizer, that allows you to tokenize a string. The class is part of the java.util package, so you need the following import statement in any program that uses it:

```
import java.util.StringTokenizer;
```

When you create an instance of the StringTokenizer class, you pass a string as an argument to one of the constructors. The tokens will be extracted from this string. [Table 8-11](#) summarizes the class's three constructors.

**Table 8-11 The StringTokenizer constructors**

| Constructor                                                 | Description                                                                                                            |
|-------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|
| <code>StringTokenizer(String str)</code>                    | The string to be tokenized is passed into str. Whitespace characters (space, tab, and newline) are used as delimiters. |
| <code>StringTokenizer(String str, String delimiters)</code> | The string to be tokenized is passed into str. The characters in delimiters will be used as delimiters.                |

```
 StringTokenizer(String
 str, String
 delimiters, Boolean
 returnDelimeters)
```

The string to be tokenized is passed into `str`. The characters in `delimiters` will be used as delimiters. If the `returnDelimeters` parameter is set to `true`, the delimiters will be included as tokens. If this parameter is set to `false`, the delimiters will not be included as tokens.

The first constructor uses whitespace characters as delimiters. The following statement instantiates a `StringTokenizer` object and uses this constructor.

```
 StringTokenizer strTokenizer = new StringTokenizer("2 4 6 8");
```

The second constructor accepts a second argument, which is a string containing one or more characters that are to be used as delimiters. The following statement creates an object using this constructor. It specifies that the `-` character is to be used as a delimiter.

```
 StringTokenizer strTokenizer = new StringTokenizer("8-14-2018", "
```

The third constructor accepts a second argument, which is a string containing one or more characters that are to be used as delimiters, and a third argument, which indicates whether the delimiters should be included as tokens. The following statement creates an object using this constructor. It specifies that the `-` character is to be used as a delimiter, and that the delimiters are to be included as tokens.

```
 StringTokenizer strTokenizer =
 new StringTokenizer("8-14-2018", "-", true);
```



## Note:

The first two constructors do not include the delimiter characters as tokens.

# Extracting Tokens

Once you have created a  `StringTokenizer` object, you can use its methods to extract tokens from the string you passed to the constructor. [Table 8-12](#) lists some of the  `StringTokenizer` methods.

**Table 8-12 Some of the StringTokenizer methods**

| Method                               | Description                                                                                                                   |
|--------------------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| <code>int countTokens()</code>       | This method returns the number of tokens left in the string.                                                                  |
| <code>boolean hasMoreTokens()</code> | This method returns <code>true</code> if there are more tokens left in the string. Otherwise, it returns <code>false</code> . |
| <code>String nextToken()</code>      | This method returns the next token found in the string.                                                                       |

The following code demonstrates how all of the tokens can be extracted from a  `StringTokenizer` object. The loop executes as long as there are tokens left to extract.

```
 StringTokenizer strTokenizer = new StringTokenizer("One Two Three"
while (strTokenizer.hasMoreTokens())
{
 System.out.println(strTokenizer.nextToken());
}
```

This code will produce the following output:

```
One
Two
Three
```

The `DateComponent` class in [Code Listing 8-8](#) uses a  `StringTokenizer`

object. Its constructor accepts a string containing a date in the form MONTH/DAY/YEAR. It extracts the month, day, and year and stores these values in the month, day, and year fields respectively. The methods getMonth, getDay, and getYear can then be used to retrieve the values. The program in [Code Listing 8-9](#) demonstrates the class.

## Code Listing 8-8 (DateComponent.java)

```
1 import java.util.StringTokenizer;
2
3 /**
4 * The DateComponent class extracts the month, day, and
5 * year from a string containing a date.
6 */
7
8 public class DateComponent
9 {
10 private String month, // To hold a month
11 day, // To hold a day
12 year; // To hold a year
13
14 /**
15 * The constructor accepts a string containing a date
16 * in the form MONTH/DAY/YEAR. It extracts the month,
17 * day, and year from the string.
18 */
19
20 public DateComponent(String dateStr)
21 {
22 // Create a StringTokenizer object. The string to
23 // tokenize is dateStr, and "/" is the delimiter.
24 StringTokenizer strTokenizer =
25 new StringTokenizer(dateStr, "/");
26
27 // Get the first token, which is the month.
28 month = strTokenizer.nextToken();
29
30 // Get the next token, which is the day.
31 day = strTokenizer.nextToken();
32
33 // Get the next token, which is the year.
```

```

34 year = strTokenizer.nextToken();
35 }
36
37 /**
38 * The getMonth method returns the month field.
39 */
40
41 public String getMonth()
42 {
43 return month;
44 }
45
46 /**
47 * The getDay method returns the day field.
48 */
49
50 public String getDay()
51 {
52 return day;
53 }
54
55 /**
56 * The getYear method returns the year field.
57 */
58
59 public String getYear()
60 {
61 return year;
62 }
63 }
```

## Code Listing 8-9 (DateTester.java)

```

1 /**
2 * This program demonstrates the DateComponent class.
3 */
4
5 public class DateTester
6 {
7 public static void main(String[] args)
8 {
9 // Create a string containing a date.
10 String date = "10/23/2018";
11
12 // Create a DateComponent object, initialized
```

```
13 // with the date.
14 DateComponent dc = new DateComponent(date);
15
16 // Display the components of the date.
17 System.out.println("Here's the date: " + date);
18 System.out.println("The month is " + dc.getMonth());
19 System.out.println("The day is " + dc.getDay());
20 System.out.println("The year is " + dc.getYear());
21 }
22 }
```

## Program Output

```
Here's the date: 10/23/2018
The month is 10
The day is 23
The year is 2018
```

# Using Multiple Delimiters

Some situations require that you use multiple characters as delimiters in the same string. For example, look at the following email address:

[joe@gaddisbooks.com](mailto:joe@gaddisbooks.com)

This string uses two delimiters: @ (the at symbol) and . (the period). To extract the tokens from this string, we must specify both characters as delimiters to the constructor. Here is an example:

```
StringTokenizer strTokenizer =
 new StringTokenizer("joe@gaddisbooks.com", "@.");
while (strTokenizer.hasMoreTokens())
{
 System.out.println(strTokenizer.nextToken());
}
```

This code will produce the following output:

```
joe
gaddisbooks
com
```

# Trimming a String Before Tokenizing

When you are tokenizing a string that was entered by the user, and you are using characters other than whitespaces as delimiters, you will probably want to trim the string before tokenizing it. Otherwise, if the user enters leading whitespace characters, they will become part of the first token. Likewise, if the user enters trailing whitespace characters, they will become part of the last token. For example, look at the following code:

```
// Create a string with leading and trailing whitespaces.
String str = " one;two;three ";
// Tokenize the string using the semicolon as a delimiter.
 StringTokenizer strTokenizer = new StringTokenizer(str, ";");
// Display the tokens.
while (strTokenizer.hasMoreTokens())
{
 System.out.println("*" + strTokenizer.nextToken() + "*");
}
```

This code will produce the following output:

```
* one*
two
*three *
```

To prevent leading and/or trailing whitespace characters from being included in the first and last tokens, use the `String` class's `trim` method to remove them. Here is the same code, modified to use the `trim` method:

```
String str = " one;two;three ";
StringTokenizer strTokenizer =
 new StringTokenizer(str.trim(), ";");
while (strTokenizer.hasMoreTokens())
{
 System.out.println("*" + strTokenizer.nextToken() + "*");
}
```

This code will produce the following output:

```
one
two
three
```

See the `SerialNumber` Class Case Study, available on this book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis), for another example using the  `StringTokenizer` class.

## The String Class's split Method

The `String` class has a method named `split` that tokenizes a string and returns an array of `String` objects. Each element in the array is one of the tokens. The following code, which is taken from the program `SplitDemo1.java` (in the book's source code), shows an example of the method's use:

```
// Create a String to tokenize.
String str = "one two three four";
// Get the tokens from the string.
String[] tokens = str.split(" ");
// Display each token.
for (String s : tokens)
 System.out.println(s);
```

The argument passed to the `split` method indicates the delimiter. In this example, a space is used as the delimiter. The code will produce the following output:

```
one
two
three
four
```

The argument that you pass to the `split` method is a *regular expression*. A regular expression is a string that specifies a pattern of characters. Regular expressions can be powerful tools, and are commonly used to search for patterns that exist in strings, files, or other collections of text. A complete discussion of regular expressions is outside the scope of this book. However, we will discuss some basic uses of regular expressions for the purpose of

tokenizing strings.

In the previous example, we passed a string containing a single space to the `split` method. This specified that the space character was the delimiter. The `split` method also allows you to use multi-character delimiters. This means that you are not limited to a single character as a delimiter. Your delimiters can be entire words, if you wish. The following code, which is taken from the program `SplitDemo2.java` (in the book's source code), demonstrates this method.

```
// Create a string to tokenize.
String str = "one and two and three and four";
// Get the tokens, using " and " as the delimiter.
String[] tokens = str.split(" and ");
// Display the tokens.
for (String s : tokens)
 System.out.println(s);
```

This code will produce the following output:

```
one
two
three
four
```

The previous code demonstrated multi-character delimiters (delimiters containing multiple characters). You can also specify a series of characters where each individual character is a delimiter. In our discussion of the  `StringTokenizer` class, we used the following string as an example requiring multiple delimiters:

[joe@gaddisbooks.com](mailto:joe@gaddisbooks.com)

This string uses two delimiters: @ (the “at” character) and . (the period). To specify that both the @ character and the . character are delimiters, we must enclose them in brackets inside our regular expression. The regular expression will look like this:

"[@.]"

Because the @ and . characters are enclosed in brackets, they will each be

considered as a delimiter. The following code, which is taken from the program *SplitDemo3.java* (in the book's source code), demonstrates this:

```
// Create a string to tokenize.
String str = "joe@gaddisbooks.com";
// Get the tokens, using @ and . as delimiters.
String[] tokens = str.split("[@.]");
// Display the tokens.
for (String s : tokens)
 System.out.println(s);
```

This code will produce the following output:

```
joe
gaddisbooks
com
```



## Checkpoint

1. 8.25 Look at the following string:

"apples pears bananas"

1. This string contains three tokens. What are they?
2. What character is the delimiter?

2. 8.26 Look at the following code.

```
 StringTokenizer st = new StringTokenizer("one two three four"
 int x = st.countTokens();
 String stuff = st.nextToken();
```

1. What value will be stored in *x*?
2. What value will the *stuff* variable reference?

3. 8.27 Look at the following string:

"/home/rjones/mydata.txt"

1. Write the declaration of a StringTokenizer object that can be used to extract the following tokens from the string: home, rjones, mydata, and txt .
2. Write code using the String class's split method that can be used to extract the same tokens specified in part a.
4. 8.28 Look at the following string:

"dog\$cat@bird%squirrel"

Write code using the String class's split method that can be used to extract the following tokens from the string: dog, cat, bird, and squirrel.

# **8.6 Wrapper Classes for the Numeric Data Types**

## **Concept:**

The Java API provides wrapper classes for each of the numeric data types. These classes have methods that perform useful operations involving primitive numeric values.

Earlier in this chapter, we discussed the character wrapper class and some of its static methods. The Java API also provides wrapper classes for all of the numeric primitive data types, as listed in [Table 8-13](#). These wrapper classes have numerous static methods that perform useful operations.

## **Table 8-13 Wrapper classes for the numeric primitive data types**

### **Wrapper Class Primitive Type It Applies To**

|         |        |
|---------|--------|
| Byte    | byte   |
| Double  | double |
| Float   | float  |
| Integer | int    |
| Long    | long   |
| Short   | short  |

# The Parse Methods

In some programming problems, a string containing a number, such as “127.89”, must be converted to a numeric data type so it can be mathematically processed. Each of the numeric wrapper classes has a static method that converts a string to a number. For example, the `Integer` class has a method that converts a string to an `int`, the `Double` class has a method that converts a string to a `double`, and so forth. These methods are known as *parse methods* because their names begin with the word “parse.” [Table 8-14](#) lists each wrapper class’s parse method.

## Table 8-14 The parse methods

| Wrapper Class        | Parse Method                                |
|----------------------|---------------------------------------------|
| <code>Byte</code>    | <code>byte parseByte(String str)</code>     |
| <code>Double</code>  | <code>double parseDouble(String str)</code> |
| <code>Float</code>   | <code>float parseFloat(String str)</code>   |
| <code>Integer</code> | <code>int parseInt(String str)</code>       |
| <code>Long</code>    | <code>long parseLong(String str)</code>     |
| <code>Short</code>   | <code>short parseShort(String str)</code>   |

The following code demonstrates how to use the parse methods.

```
byte bVar = Byte.parseByte("1"); // Store 1 in bVar.
int iVar = Integer.parseInt("2599"); // Store 2599 in iVar.
short sVar = Short.parseShort("10"); // Store 10 in sVar.
long lVar = Long.parseLong("15908"); // Store 15908 in lV
float fVar = Float.parseFloat("12.3"); // Store 12.3 in fVa
double dVar = Double.parseDouble("7945.6"); // Store 7945.6 in d
```

Of course, you can pass `String` objects as arguments to these methods too, as shown here:

```
String str = "2599";
int iVar = Integer.parseInt(str); // Store 2599 in iVar.
```

# The Static `toString` Methods

Each of the numeric wrapper classes has a static `toString` method that converts a number to a string. The method accepts the number as its argument and returns a string representation of that number. The following code demonstrates this method:

```
int i = 12;
double d = 14.95;
String str1 = Integer.toString(i);
String str2 = Double.toString(d);
```

# The `toBinaryString`, `toHexString`, and `toOctalString` Methods

The `toBinaryString`, `toHexString`, and `toOctalString` methods are static members of the `Integer` and `Long` wrapper classes. These methods accept an integer as an argument and return a string representation of that number converted to binary, hexadecimal, or octal. The following code demonstrates these methods:

```
int number = 14;
System.out.println(Integer.toBinaryString(number));
System.out.println(Integer.toHexString(number));
System.out.println(Integer.toOctalString(number));
```

This code will produce the following output:

```
1110
e
16
```

# The `MIN_VALUE` and `MAX_VALUE` Constants

The numeric wrapper classes each have a set of static final variables named `MIN_VALUE` and `MAX_VALUE`. These variables hold the minimum and maximum values for a particular data type. For example, `Integer.MAX_VALUE` holds the maximum value that an `int` can hold. The following code displays the minimum and maximum values for an `int`:

```
System.out.println("The minimum value for an " +
 "int is " + Integer.MIN_VALUE);
System.out.println("The maximum value for an " +
 "int is " + Integer.MAX_VALUE);
```

## Autoboxing and Unboxing

It is possible to create objects from the wrapper classes. One way is to pass an initial value to the constructor, as shown here:

```
Integer number = new Integer(7);
```

This creates an `Integer` object initialized with the value 7, referenced by the variable `number`. Another way is to simply declare a wrapper class variable, then assign a primitive value to it. For example, look at the following code:

```
Integer number;
number = 7;
```

The first statement in this code declares an `Integer` variable named `number`. It does not create an `Integer` object, just a variable. The second statement is a simple assignment statement. It assigns the primitive value 7 to the variable. You might suspect that this will cause an error. After all, `number` is a reference variable, not a primitive variable. However, because `number` is a wrapper class variable, Java performs an autoboxing operation. *Autoboxing* is Java's process of automatically “boxing up” a value inside an object. When this assignment statement executes, Java boxes up the value 7 inside an `Integer` object, then assigns the address of that object to the `number` variable.

*Unboxing* is the opposite of boxing. It is the process of converting a wrapper class object to a primitive type. The following code demonstrates an unboxing operation:

```
Integer myInt = 5; // Autoboxes the value 5
int primitiveNumber;
primitiveNumber = myInt; // Unboxes the object
```

The first statement in this code declares `myInt` as an `Integer` reference variable. The primitive value 5 is autoboxed, and the address of the resulting object is assigned to the `myInt` variable. The second statement declares `primitiveNumber` as an `int` variable. The third statement assigns the `myInt` object to `primitiveNumber`. When this statement executes, Java automatically unboxes the `myInt` wrapper class object and stores the resulting value, which is 5, in `primitiveNumber`.

Although you rarely need to create an instance of a wrapper class, Java's autoboxing and unboxing features make some operations more convenient. Occasionally, you will find yourself in a situation where you want to perform an operation using a primitive variable, but the operation can be used only with an object. For example, recall the `ArrayList` class we discussed in [Chapter 7](#). An `ArrayList` is an array-like object that can be used to store other objects. You cannot, however, store primitive values in an `ArrayList`. It is intended for objects only. If you compile the following statement, an error will occur:

```
ArrayList<int> list = new ArrayList<int>(); // ERROR!
```

However, you can store wrapper class objects in an `ArrayList`. If we need to store `int` values in an `ArrayList`, we have to specify that the `ArrayList` will hold `Integer` objects. Here is an example:

```
ArrayList<Integer> list = new ArrayList<Integer>(); // Okay.
```

This statement declares that `list` references an `ArrayList` that can hold `Integer` objects. One way to store an `int` value in the `ArrayList` is to instantiate an `Integer` object, initialize it with the desired `int` value, then pass the `Integer` object to the `ArrayList`'s `add` method. Here is an example.

```
ArrayList<Integer> list = new ArrayList<Integer>();
Integer myInt = 5;
list.add(myInt);
```

However, Java's autoboxing and unboxing features make it unnecessary to

create the `Integer` object. If you add an `int` value to the `ArrayList`, Java will autobox the value. The following code works without any problems:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
```

When the value 5 is passed to the `add` method, Java boxes the value up in an `Integer` object. When necessary, Java also unboxes values that are retrieved from the `ArrayList`. The following code demonstrates this:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
int primitiveNumber = list.get(0);
```

The last statement in this code retrieves the item at index 0. Because the item is being assigned to an `int` variable, Java unboxes it and stores the primitive value in the `int` variable.



## Checkpoint

1. 8.29 Write a statement that converts the following string to a double and stores it in the double variable number:

```
String str = "894.56";
```

2. 8.30 Write a statement that converts the following integer to a string and stores it in the String object referenced by str:

```
int i = 99;
```

3. 8.31 What wrapper class methods convert a number from decimal to another numbering system? What wrapper classes are these methods a member of?

4. 8.32 What is the purpose of the `MIN_VALUE` and `MAX_VALUE` variables that are members of the numeric wrapper classes?

## 8.7 Focus on Problem Solving: The TestScoreReader Class

Professor Harrison keeps her students' test scores in a Microsoft Excel spreadsheet. [Figure 8-3](#) shows a set of five test scores for five students. Each column holds a test score, and each row represents the scores for one student.

### Figure 8-3 Microsoft Excel spreadsheet

|   | A  | B  | C  | D  | E  | F |
|---|----|----|----|----|----|---|
| 1 | 87 | 79 | 91 | 82 | 94 |   |
| 2 | 72 | 79 | 81 | 74 | 88 |   |
| 3 | 94 | 92 | 81 | 89 | 96 |   |
| 4 | 77 | 56 | 67 | 81 | 79 |   |
| 5 | 79 | 82 | 85 | 81 | 90 |   |
| 6 |    |    |    |    |    |   |

[Figure 8-3 Full Alternative Text](#)

In addition to manipulating the scores in Excel, Dr. Harrison wants to write a Java application that accesses them. Excel, like many commercial applications, has the ability to export data to a text file. When the data in a spreadsheet is exported, each row is written to a line, and the values in the cells are separated by commas. For example, when the data shown in [Figure 8-3](#) is exported, it will be written to a text file in the following format:

```
87,79,91,82,94
72,79,81,74,88
94,92,81,89,96
77,56,67,81,79
79,82,85,81,90
```

This is called the *comma separated value* file format. When you save a

spreadsheet in this format, Excel saves it to a file with the .csv extension. Dr. Harrison decides to export her spreadsheet to a .csv file, then write a Java program that reads the file. The program will use the String class's split method to extract the test scores from each line, and a wrapper class to convert the tokens to numeric values. As an experiment, she writes the TestScoreReader class shown in [Code Listing 8-10](#).

## Code Listing 8-10 (TestScoreReader.java)

```
1 import java.util.Scanner; // For Scanner
2 import java.io.*; // For File and IOException
3
4 /**
5 * The TestScoreReader class reads test scores as
6 * tokens from a file and calculates the average
7 * of each line of scores.
8 */
9
10 public class TestScoreReader
11 {
12 private Scanner inputFile;
13 private String line;
14
15 /**
16 * The constructor opens a file to read
17 * the grades from.
18 */
19
20 public TestScoreReader(String filename)
21 throws IOException
22 {
23 File file = new File(filename);
24 inputFile = new Scanner(file);
25 }
26
27 /**
28 * The readNextLine method reads the next line
29 * from the file.
30 */
31
32 public boolean readNextLine() throws IOException
```

```
33 {
34 boolean lineRead; // Flag variable
35
36 // Determine whether there is more to read.
37 lineRead = inputFile.hasNext();
38
39 // If so, read the next line.
40 if (lineRead)
41 line = inputFile.nextLine();
42
43 return lineRead;
44 }
45
46 /**
47 * The getAverage method calculates the average
48 * of the last set of test scores read from the file.
49 */
50
51 public double getAverage()
52 {
53 int total = 0; // Accumulator
54 double average; // The average test score
55
56 // Tokenize the last line read from the file.
57 String[] tokens = line.split(",");
58
59 // Calculate the total of the test scores.
60 for (String str : tokens)
61 {
62 total += Integer.parseInt(str);
63 }
64
65 // Calculate the average of the scores.
66 // Use a cast to avoid integer division.
67 average = (double) total / tokens.length;
68
69 // Return the average.
70 return average;
71 }
72
73 /**
74 * The close method closes the file.
75 */
76
77 public void close() throws IOException
78 {
79 inputFile.close();
80 }
```

```
81 }
```

The constructor accepts the name of a file as an argument and opens the file. The `readNextLine` method reads a line from the file and stores it in the `line` field. The method returns `true` if a line was successfully read from the file, or `false` if there are no more lines to read. The `getAverage` method tokenizes the last line read from the file, converts the tokens to `double` values, and calculates the average of the values. The average is returned. The program in [Code Listing 8-11](#) uses the `TestScoreReader` class to open the file `Grades.csv` and get the averages of the test scores it contains.

## Code Listing 8-11 (`TestAverages.java`)

```
1 import java.io.*; // Needed for IOException
2
3 /**
4 * This program uses the TestScoreReader class to read
5 * test scores from a file and get their averages.
6 */
7
8 public class TestAverages
9 {
10 public static void main(String[] args)
11 throws IOException
12 {
13 double average; // To hold an average
14 int studentNumber = 1; // To count students
15
16 // Create a TestScoreReader object.
17 TestScoreReader scoreReader =
18 new TestScoreReader("Grades.csv");
19
20 // Process the file contents.
21 while (scoreReader.readNextLine())
22 {
23 // Get this student's average.
24 average = scoreReader.getAverage();
25
26 // Display this student's average.
27 System.out.println("Average for student number " +
```

```
28 studentNumber + " is " +
29 average);
30
31 // Increment the student number.
32 studentNumber++;
33 }
34
35 // Close the file.
36 scoreReader.close();
37 System.out.println("No more scores.");
38 }
39 }
```

## Program Output

```
Average for student number 1 is 86.6
Average for student number 2 is 78.8
Average for student number 3 is 90.4
Average for student number 4 is 72.0
Average for student number 5 is 83.4
No more scores.
```

Dr. Harrison's class works properly, and she decides that she can expand it to perform other, more complex operations.

## 8.8 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- Using static wrapper class methods as if they were instance methods. Many of the most useful wrapper class methods are static, and you should call them directly from the class.
- Trying to use `String` comparison methods such as `startsWith` and `endsWith` for case-insensitive comparisons. Most of the `String` comparison methods are case sensitive. Only the `regionMatches` method performs a case-insensitive comparison.
- Thinking of the first position of a string as 1. Many of the `String` and `StringBuilder` methods accept a character position within a string as an argument. Remember, the position numbers in a string start at zero. If you think of the first position in a string as 1, you will cause an off-by-one error.
- Thinking of the ending position of a substring as part of the substring. Methods such as `getChars` accept the starting and ending position of a substring as arguments. The character at the *start* position is included in the substring, but the character at the *end* position is not included. (The last character in the substring ends at *end* –1.)
- Extracting more tokens from a  `StringTokenizer` object than exist. Trying to extract more tokens than exist from a  `StringTokenizer` object will cause an error. You can use the `countTokens` method to determine the number of tokens and the `hasMoreTokens` method to determine whether there are any more unread tokens.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. The `isDigit`, `isLetter`, and `isLetterOrDigit` methods are members of this class.
  1. `String`
  2. `Char`
  3. `Character`
  4. `StringBuilder`
2. This method converts a character to uppercase.
  1. `makeUpperCase`
  2. `toUpperCase`
  3. `isUpperCase`
  4. `upperCase`
3. The `startsWith`, `endsWith`, and `regionMatches` methods are members of this class.
  1. `String`
  2. `Char`
  3. `Character`
  4.  `StringTokenizer`

4. The `indexOf` and `lastIndexOf` methods are members of this class.
  1. `String`
  2. `Integer`
  3. `Character`
  4.  `StringTokenizer`
5. The `substring`, `getChars`, and `toCharArray` methods are members of this class.
  1. `String`
  2. `Float`
  3. `Character`
  4.  `StringTokenizer`
6. This `String` class method performs the same operation as the `+` operator when used on strings.
  1. `add`
  2. `join`
  3. `concat`
  4. `plus`
7. The `String` class has several overloaded versions of a method that accepts a value of any primitive data type as its argument and returns a string representation of the value. The name of the method is
  1. `stringValue`

2. `valueOf`
  3. `getString`
  4. `valToString`
8. If you do not pass an argument to the `StringBuilder` constructor, the object will have enough memory to store this many characters.
  1. 16
  2. 1
  3. 256
  4. Unlimited
9. This is one of the methods that are common to both the `String` and `StringBuilder` classes.
  - a.
  1. `append`
  2. `insert`
  3. `delete`
  4. `length`
10. To change the value of a specific character in a `StringBuilder` object, use this method.
  1. `changeCharAt`
  2. `setCharAt`
  3. `setChar`

4. change
11. To delete a specific character in a `StringBuilder` object, use this method.
1. `deleteCharAt`
  2. `removeCharAt`
  3. `removeChar`
  4. `expunge`
12. The character that separates tokens in a string is known as a .
1. separator
  2. tokenizer
  3. delimiter
  4. terminator
13. This  `StringTokenizer` method returns true if there are more tokens to be extracted from a string.
1. `moreTokens`
  2. `tokensLeft`
  3. `getToken`
  4. `hasMoreTokens`
14. Each of the numeric wrapper classes has a static method that converts a string to a number. All of these methods begin with this word.
1. `convert`

2. `toString`
  3. `parse`
  4. `toNumber`
15. These static `final` variables are members of the numeric wrapper classes and hold the minimum and maximum values for a particular data type.
1. `MIN_VALUE` and `MAX_VALUE`
  2. `MIN` and `MAX`
  3. `MINIMUM` and `MAXIMUM`
  4. `LOWEST` and `HIGHEST`
16. True or False: Character testing methods, such as `isLetter`, accept strings as arguments and test each character in the string.
17. True or False: If the `toUpperCase` method's argument is already uppercase, it is returned as is, with no changes.
18. True or False: If the `toLowerCase` method's argument is already lowercase, it will be inadvertently converted to uppercase.
19. True or False: The `startsWith` and `endsWith` methods are case sensitive.
20. True or False: There are two versions of the `regionMatches` method: one that is case sensitive, and one that can be case insensitive.
21. True or False: The `indexOf` and `lastIndexOf` methods find characters, but cannot find substrings.
22. True or False: The `String` class's `replace` method can replace individual characters, but not substrings.

23. True or False: The `StringBuilder` class's `replace` method can replace individual characters, but not substrings.
24. True or False: You can use the `=` operator to assign a string to a `StringBuilder` object.
25. True or False: To get the value of a wrapper class object, you must call a method.

## Find the Error

Find the error in each of the following code segments.

1. 

```
int number = 99;
String str;
// Convert number to a string.
str.valueOf(number);
```
2. 

```
// Store a name in a StringBuilder object.
StringBuilder name = "Joe Schmoe";
```
3. 

```
int number;
String str = "99";
// Convert str to an int.
number = str.parseInt();
```
4. 

```
// Change the very first character of a
// StringBuilder object to 'Z'.
str.setCharAt(1, 'Z');
```
5. 

```
// Tokenize a string that is delimited
// with semicolons. The string has three tokens.
 StringTokenizer strTokenizer =
 new StringTokenizer("One;Two;Three");
// Extract the three tokens from the string.
while (strTokenizer.hasMoreTokens())
{
 System.out.println(strTokenizer.nextToken());
}
```

# Algorithm Workbench

1. The following `if` statement determines whether `choice` is equal to ‘Y’ or ‘y’:

```
if (choice == 'Y' || choice == 'y')
```

Rewrite this statement so that it makes only one comparison and does not use the `||` operator. (Hint: Use either the `toUpperCase` or `toLowerCase` methods.)

2. Write a loop that counts the number of space characters that appear in the `String` object `str`.
3. Write a loop that counts the number of digits that appear in the `String` object `str`.
4. Write a loop that counts the number of lowercase characters that appear in the `String` object `str`.
5. Write a method that accepts a `String` object as an argument and returns `true` if the argument ends with the substring “.com”. Otherwise, the method should return `false`.
6. Modify the method you wrote for question 5 so it performs a case-insensitive test. The method should return `true` if the argument ends with “.com” in any possible combination of upper and lowercase letters.
7. Write a method that accepts a `StringBuilder` object as an argument and converts all occurrences of the lowercase letter “t” in the object to uppercase.
8. Look at the following string:

“cookies>milk>fudge:cake:ice cream”

1. Write code using a  `StringTokenizer` object that extracts the following tokens from the string and displays them: `cookies`, `milk`,

fudge, cake, and ice cream.

2. Write code using the `String` class's `split` method that extracts the same tokens as the code you wrote for part a.
9. Assume that `d` is a double variable. Write an `if` statement that assigns `d` to the `int` variable `i`, if the value in `d` is not larger than the maximum value for an `int`.
10. Write code that displays the contents of the `int` variable `i` in binary, hexadecimal, and octal.
11. Look at the following declaration statements:

```
String str = "237.89";
double value;
```

Write a statement that converts the string referenced by `str` to a double and stores the result in `value`.

## Short Answer

1. Why should you use `StringBuilder` objects instead of `String` objects in a program that makes lots of changes to strings?
2. A program reads a string as input from the user for the purpose of tokenizing it. Why is it a good idea to trim the string before tokenizing it?
3. Each of the numeric wrapper classes has a “parse” method. What do these methods do?
4. Each of the numeric wrapper classes has a static `toString` method. What do these methods do?
5. How can you determine the minimum and maximum values that can be stored in a variable of a given data type?

# Programming Challenges

## 1. Backward String

Write a method that accepts a `String` object as an argument and displays its contents backward. For instance, if the string argument is “gravity”, the method should display “ytivarg”. Demonstrate the method in a program that asks the user to input a string, then passes it to the method.

## 2. Word Counter

Write a method that accepts a `String` object as an argument and returns the number of words it contains. For instance, if the argument is “Four score and seven years ago” the method should return the number 6. Demonstrate the method in a program that asks the user to input a string then passes it to the method. The number of words in the string should be displayed on the screen.

## 3. Sentence Capitalizer

Write a method that accepts a `String` object as an argument and returns a copy of the string with the first character of each sentence capitalized. For instance, if the argument is “hello. my name is Joe. what is your name?” the method should return the string, “Hello. My name is Joe. What is your name?” Demonstrate the method in a program that asks the user to input a string then passes it to the method. The modified string should be displayed on the screen.



**VideoNote** The Sentence Capitalizer Problem

## 4. Vowels and Consonants

Write a class with a constructor that accepts a `String` object as its argument. The class should have a method that returns the number of

vowels in the string, and another method that returns the number of consonants in the string. Demonstrate the class in a program that performs the following steps:

1. The user is asked to enter a string.
  2. The program displays the following menu:
    1. Count the number of vowels in the string
    2. Count the number of consonants in the string
    3. Count both the vowels and consonants in the string
    4. Enter another string
    5. Exit the program
  3. The program performs the operation selected by the user and repeats until the user selects e to exit the program.
5. Password Verifier

Imagine you are developing a software package for an online shopping site that requires users to enter their own passwords. Your software requires that users' passwords meet the following criteria:

- The password should be at least six characters long.
- The password should contain at least one uppercase and at least one lowercase letter.
- The password should have at least one digit.

Write a class that verifies that a password meets the stated criteria. Demonstrate the class in a program that allows the user to enter a password, then displays a message indicating whether it is valid or not.

6. Telemarketing Phone Number List

Write a program that has two parallel arrays of `String` objects. One of the arrays should hold people's names, and the other should hold their phone numbers. Here are sample contents of both arrays.

| <b>name Array Sample Contents</b> | <b>phone Array Sample Contents</b> |
|-----------------------------------|------------------------------------|
| "Harrison, Rose"                  | "555-2234"                         |
| "James, Jean"                     | "555-9098"                         |
| "Smith, William"                  | "555-1785"                         |
| "Smith, Brad"                     | "555-9224"                         |

The program should ask the user to enter a name or the first few characters of a name to search for in the array. The program should display all of the names that match the user's input and their corresponding phone numbers. For example, if the user enters "Smith," the program should display the following names and phone numbers from the list:

Smith, William: 555-1785  
Smith, Brad: 555-9224

## 7. Check Writer

Write a program that displays a simulated paycheck. The program should ask the user to enter the date, the payee's name, and the amount of the check. It should then display a simulated check with the dollar amount spelled out, as shown here:

Date: 11/24/2018  
Pay to the Order of: John Phillips \$1920.85  
One thousand nine hundred twenty and 85 cents

## 8. Sum of Numbers in a String

Write a program that asks the user to enter a series of numbers separated by commas. Here is an example of valid input:

7, 9, 10, 2, 18, 6

The program should calculate and display the sum of all the numbers.

## 9. Sum of Digits in a String

Write a program that asks the user to enter a series of single digit numbers with nothing separating them. The program should display the sum of all the single digit numbers in the string. For example, if the user enters 2514, the method should return 12, which is the sum of 2, 5, 1, and 4. The program should also display the highest and lowest digits in the string. (Hint: Convert the string to an array.)

## 10. Word Counter

Write a program that asks the user for the name of a file. The program should display the number of words that the file contains.

## 11. Sales Analysis

If you have downloaded the book's source code from [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis), you will find the file *SalesData.txt* in the *Chapter 08* folder. This file contains the dollar amount of sales that a retail store made each day for a number of weeks. Each line in the file contains seven numbers, which are the sales numbers for one week. The numbers are separated by a comma. The following line is an example from the file:

1245.67, 1490.07, 1679.87, 2378.46, 1783.92, 1468.99, 2059.77

Write a program that opens the file and processes its contents. The program should display the following:

- The total sales for each week
- The average daily sales for each week
- The total sales for all of the weeks
- The average weekly sales

- The week number that had the highest amount of sales
- The week number that had the lowest amount of sales

## 12. Miscellaneous String Operations

Write a class with the following static methods:

**wordCount.** This method should accept a `String` object as an argument and return the number of words contained in the object.

**arrayToString.** This method accepts a `char` array as an argument and converts it to a `String` object. The method should return a reference to the `String` object.

**mostFrequent.** This method accepts a `String` object as an argument and returns the character that occurs most frequently in the object.

**replaceSubstring.** This method accepts three `String` objects as arguments. Let's call them `string1`, `string2`, and `string3`. It searches `string1` for all occurrences of `string2`. When it finds an occurrence of `string2`, it replaces it with `string3`. For example, suppose the three arguments have the following values:

```
string1: "the dog jumped over the fence"
string2: "the"
string3: "that"
```

With these three arguments, the method would return a `String` object with the value “that dog jumped over that fence”.

Demonstrate each of these methods in a complete program.

## 13. Alphabetic Telephone Number Translator

Many companies use telephone numbers such as 555-GET-FOOD so that the number is easier for their customers to remember. On a standard telephone, the alphabetic letters are mapped to numbers in the following fashion:

- A, B, and C = 2
- D, E, and F = 3
- G, H, and I = 4
- J, K, and L = 5
- M, N, and O = 6
- P, Q, R, and S = 7
- T, U, and V = 8
- W, X, Y, and Z = 9

Write an application that asks the user to enter a 10-character telephone number in the format XXX-XXX-XXXX. The application should display the telephone number with any alphabetic characters that appeared in the original translated to their numeric equivalent. For example, if the user enters 555-GET-FOOD, the application should display 555-438-3663.

#### 14. Word Separator

Write a program that accepts as input a sentence in which all the words are run together, but the first character of each word is uppercase. Convert the sentence to a string in which the words are separated by spaces, and only the first word starts with an uppercase letter. For example the string “StopAndSmellTheRoses.” would be converted to “Stop and smell the roses.”

#### 15. Pig Latin

Write a program that reads a sentence as input and converts each word to “Pig Latin”. In one version of Pig Latin, you convert a word by removing the first letter, placing that letter at the end of the word, and then appending “ay” to the word. Here is an example:

English: I SLEPT MOST OF THE NIGHT

Pig Latin: IAY LEPTSAY OSTMAY FOAY HETAY IGHTNAY

## 16. Morse Code Converter

Write a program that asks the user to enter a string, then converts that string to Morse code. Morse code is a code where each letter of the English alphabet, each digit, and various punctuation characters are represented by a series of dots and dashes. [Table 8-15](#) shows part of the code.

## Table 8-15 Morse code

| Character        | Code      | Character | Code        | Character | Code      | Character | Code |
|------------------|-----------|-----------|-------------|-----------|-----------|-----------|------|
| space            | space     | 6         | - . . .     | G         | - - .     | Q         | -    |
| comma            | - - . - - | 7         | - - . . .   | H         | . . . .   | R         | .    |
| period           | - - - . - | 8         | - - - - .   | I         | ..        | S         | .    |
| question<br>mark | - - - - . | 9         | - - - - - . | J         | . - - -   | T         | -    |
| 0                | - - - - - | A         | . -         | K         | - . -     | U         | .    |
| 1                | - - - - - | B         | - . .       | L         | . - - .   | V         | .    |
| 2                | - - - - - | C         | - . . .     | M         | - - .     | W         | .    |
| 3                | - - - - - | D         | - . . . .   | N         | - . .     | X         | -    |
| 4                | - - - - - | E         | - . . . . . | O         | - - - .   | Y         | -    |
| 5                | - - - - - | F         | - . . . . . | P         | - - - . . | Z         | -    |

## 17. Lottery Statistics

To play the PowerBall lottery, you buy a ticket that has five numbers in the range of 1–69, and a “PowerBall” number in the range of 1–26. (You can pick the numbers yourself, or you can let the ticket machine randomly pick them for you.) Then, on a specified date, a winning set of numbers are randomly selected by a machine. If your first five numbers match the first five winning numbers in any order, and your PowerBall number matches the winning PowerBall number, then you win the

jackpot, which is a very large amount of money. If your numbers match only some of the winning numbers, you win a lesser amount, depending on how many of the winning numbers you have matched.

In the student sample programs for this chapter, you will find a file named pbnumbers.txt, containing the winning lottery numbers that were selected between February 3, 2010 and May 11, 2016 (the file contains 654 sets of winning numbers). Here is an example of the first few lines of the file's contents:

```
17 22 36 37 52 24
14 22 52 54 59 04
05 08 29 37 38 34
10 14 30 40 51 01
07 08 19 26 36 15
and so on . . .
```

Each line in the file contains the set of six numbers that were selected on a given date. The numbers are separated by a space, and the last number in each line is the PowerBall number for that day. For example, the first line in the file shows the numbers for February 3, 2010, which are 17, 22, 36, 37, 52, and the PowerBall number 24.

Write one or more programs that work with this file to perform the following:

- Display the 10 most common numbers, ordered by frequency
- Display the 10 least common numbers, ordered by frequency
- Display the 10 most overdue numbers (numbers that haven't been drawn in a long time), ordered from most overdue to least overdue
- Display the frequency of each number 1-69, and the frequency of each Powerball number 1-26

## 18. Gas Prices

In the student sample program files for this chapter, you will find a text file named GasPrices.txt. The file contains the weekly average prices for

a gallon of gas in the US, beginning on April 5<sup>th</sup>, 1993, and ending on August 26<sup>th</sup>, 2013. Here is an example of the first few lines of the file's contents:

```
04-05-1993:1.068
04-12-1993:1.079
04-19-1993:1.079
04-26-1993:1.086
05-03-1993:1.086
and so on . . .
```

Each line in the file contains the average price for a gallon of gas on a specific date. Each line is formatted in the following way:

*MM-DD-YYYY:Price*

*MM* is the two-digit month, *DD* is the two-digit day, and *YYYY* is the four-digit year. *Price* is the average price per gallon of gas on the specified date.

For this assignment you are to write one or more programs that read the contents of the file and perform the following calculations:

- Average Price Per Year: Calculate the average price of gas per year, for each year in the file. The file's data starts in April of 1993, and ends in August, 2013. (For the years 1993 and 2013, use only the data that is present for those years.)
- Average Price Per Month: Calculate the average price for each month in the file.
- Highest and Lowest Prices Per Year: For each year in the file, determine the date and amount for the lowest price, and the highest price.
- List of Prices, Lowest to Highest: Generate a text file that lists the dates and prices, sorted from the lowest price to the highest.
- List of Prices, Highest to lowest: Generate a text file that lists the dates and prices, sorted from the highest price to the lowest.

You can write one program to perform all of these calculations, or you can write different programs, one for each calculation.

# Chapter 9 Inheritance

## Topics

1. [9.1 What Is Inheritance?](#)
2. [9.2 Calling the Superclass Constructor](#)
3. [9.3 Overriding Superclass Methods](#)
4. [9.4 Protected Members](#)
5. [9.5 Classes That Inherit from Subclasses](#)
6. [9.6 The Object Class](#)
7. [9.7 Polymorphism](#)
8. [9.8 Abstract Classes and Abstract Methods](#)
9. [9.9 Interfaces](#)
10. [9.10 Anonymous Classes](#)
11. [9.11 Functional Interfaces and Lambda Expressions](#)
12. [9.12 Common Errors to Avoid](#)

# **9.1 What Is Inheritance?**

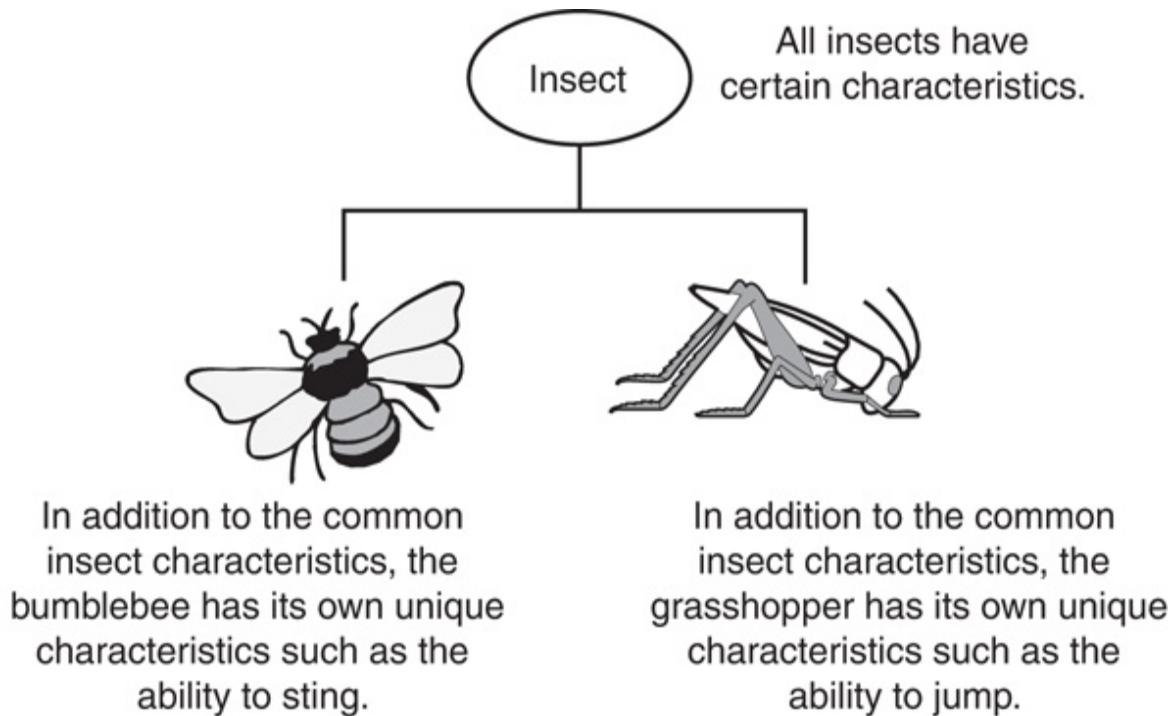
## **Concept:**

**Inheritance allows a new class to be based on an existing class. The new class inherits the members of the class it is based on.**

## **Generalization and Specialization**

In the real world, you can find many objects that are specialized versions of other more general objects. For example, the term “insect” describes a very general type of creature with numerous characteristics. Because grasshoppers and bumblebees are insects, they have all the general characteristics of an insect. In addition, they have special characteristics of their own. For example, the grasshopper has its jumping ability, and the bumblebee has its stinger. Grasshoppers and bumblebees are specialized versions of an insect. This is illustrated in [Figure 9-1](#).

**Figure 9-1 Bumblebees and grasshoppers are specialized versions of an insect**



[Figure 9-1 Full Alternative Text](#)



[VideoNote](#) Inheritance

## Inheritance and the “Is-a” Relationship

When one object is a specialized version of another object, there is an “*is-a*” relationship between them. For example, a grasshopper *is an* insect. Here are a few other examples of the “*is-a*” relationship:

- A poodle *is a* dog.
- A car *is a* vehicle.
- A flower *is a* plant.

- A rectangle *is a* shape.
- A football player *is an* athlete.

When an “is-a” relationship exists between objects, it means that the specialized object has all of the characteristics of the general object, plus additional characteristics that make it special. In object-oriented programming, *inheritance* is used to create an “is-a” relationship among classes. This allows you to extend the capabilities of a class by creating another class that is a specialized version of it.

Inheritance involves a superclass and a subclass. The *superclass* is the general class, and the *subclass* is the specialized class. (Superclasses are also called *base classes*, and subclasses are also called *derived classes*.) You can think of the subclass as an extended version of the superclass. The subclass inherits fields and methods from the superclass without any of them being rewritten. Furthermore, new fields and methods can be added to the subclass to make it more specialized than the superclass.

Let’s look at an example of how inheritance can be used. Most teachers assign various graded activities for their students to complete. A graded activity can be given a numeric score such as 70, 85, 90, and so on, and a letter grade such as A, B, C, D or F. [Figure 9-2](#) shows a UML diagram for the `GradedActivity` class, which is designed to hold the numeric score of a graded activity. The `setScore` method sets a numeric score, and the `getScore` method returns the numeric score. The `getGrade` method returns the letter grade that corresponds to the numeric score. Notice that the class does not have a programmer-defined constructor, so Java will automatically generate a default constructor for it. This will be a point of discussion later. [Code Listing 9-1](#) shows the code for the class, and the program in [Code Listing 9-2](#) demonstrates the class.

## Figure 9-2 UML diagram for the `GradedActivity` class

|                               |
|-------------------------------|
| GradedActivity                |
| - score : double              |
| + setScore(s : double) : void |
| + getScore() : double         |
| + getGrade() : char           |

[Figure 9-2 Full Alternative Text](#)

## Code Listing 9-1 (GradedActivity.java)

```

1 /**
2 * A class that holds a grade for a graded activity.
3 */
4
5 public class GradedActivity
6 {
7 private double score; // Numeric score
8
9 /**
10 * The setScore method stores its argument in
11 * the score field.
12 */
13
14 public void setScore(double s)
15 {
16 score = s;
17 }
18
19 /**
20 * The getScore method returns the score field.
21 */
22
23 public double getScore()
24 {
25 return score;
26 }
27
28 /**

```

```
29 * The getGrade method returns a letter grade
30 * determined from the score field.
31 */
32
33 public char getGrade()
34 {
35 char letterGrade; // To hold the grade
36
37 if (score >= 90)
38 letterGrade = 'A';
39 else if (score >= 80)
40 letterGrade = 'B';
41 else if (score >= 70)
42 letterGrade = 'C';
43 else if (score >= 60)
44 letterGrade = 'D';
45 else
46 letterGrade = 'F';
47
48 return letterGrade;
49 }
50 }
```

## Code Listing9-2 (GradeDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the GradedActivity class.
5 */
6
7 public class GradeDemo
8 {
9 public static void main(String[] args)
10 {
11 double testScore; // To hold a test score
12
13 // Create a Scanner object for keyboard input.
14 Scanner keyboard = new Scanner(System.in);
15
16 // Create a GradedActivity object.
17 GradedActivity grade = new GradedActivity();
18
19 // Get a test score from the user.
```

```
20 System.out.print("Enter a numeric test score: ");
21 testScore = keyboard.nextDouble();
22
23 // Set the GradedActivity object's score.
24 grade.setScore(testScore);
25
26 // Display the letter grade for that score.
27 System.out.println("The grade for that test is " +
28 grade.getGrade());
29 }
30 }
```

## Program Output with Example Input Shown in Bold

Enter a numeric test score: **89**   
The grade for that test is B

## Program Output with Example Input Shown in Bold

Enter a numeric test score: **75**   
The grade for that test is C

The `GradedActivity` class represents the general characteristics of a student's graded activity. Many different types of graded activities exist, however, such as quizzes, midterm exams, final exams, lab reports, essays, and so on. Because the numeric scores might be determined differently for each of these graded activities, we can create subclasses to handle each one. For example, a `FinalExam` class could inherit from the `GradedActivity` class. [Figure 9-3](#) shows the UML diagram for such a class, and [Code Listing 9-3](#) shows its code. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

## Figure 9-3 UML diagram for the `FinalExam` class

| FinalExam                                     |
|-----------------------------------------------|
| - numQuestions : int                          |
| - pointsEach : double                         |
| - numMissed : int                             |
| + FinalExam(questions : int,<br>missed : int) |
| + getPointsEach() : double                    |
| + getNumMissed() : int                        |

[Figure 9-3 Full Alternative Text](#)

## Code Listing 9-3 (FinalExam.java)

```

1 /**
2 * This class determines the grade for a final exam.
3 */
4
5 public class FinalExam extends GradedActivity
6 {
7 private int numQuestions; // Number of questions
8 private double pointsEach; // Points for each question
9 private int numMissed; // Number of questions missed
10
11 /**
12 * The constructor accepts as arguments the number
13 * of questions on the exam and the number of
14 * questions the student missed.
15 */
16
17 public FinalExam(int questions, int missed)
18 {
19 double numericScore; // To calculate the numeric score
20
21 // Set the numQuestions and numMissed fields.
22 numQuestions = questions;
23 numMissed = missed;
24
25 // Calculate the points for each question and
26 // the numeric score for this exam.
27 pointsEach = 100.0 / questions;

```

```

28 numericScore = 100.0 - (missed * pointsEach);
29
30 // Call the superclass's setScore method to
31 // set the numeric score.
32 setScore(numericScore);
33 }
34
35 /**
36 * The getPointsEach method returns the pointsEach
37 * field.
38 */
39
40 public double getPointsEach()
41 {
42 return pointsEach;
43 }
44
45 /**
46 * The getNumMissed method returns the numMissed
47 * field.
48 */
49
50 public int getNumMissed()
51 {
52 return numMissed;
53 }
54 }
```

The only new notation in this class declaration is in the class header (in line 5), which is shown in [Figure 9-4](#).

## Figure 9-4 First line of the FinalExam class declaration

```
public class FinalExam extends GradedActivity
```

↑  
Class being declared  
(the subclass)

↑  
Superclass

The `extends` key word indicates that this class inherits from another class (a superclass). The name of the superclass is listed after the word `extends`. So, this line of code indicates that `FinalExam` is the name of the class being declared, and `GradedActivity` is the name of the superclass it inherits from.

As you read the line, it communicates the fact that the `FinalExam` class extends the `GradedActivity` class. This makes sense because a subclass is an extension of its superclass. If we want to express the relationship between the two classes, we can say that a `FinalExam` is a `GradedActivity`.

Because the `FinalExam` class inherits from the `GradedActivity` class, it inherits all of the public members of the `GradedActivity` class. Here is a list of the members of the `FinalExam` class:

## Fields:

```
int numQuestions; Declared in FinalExam
double pointsEach; Declared in FinalExam
int numMissed; Declared in FinalExam
```

## Methods:

```
Constructor Declared in FinalExam
getPointsEach Declared in FinalExam
getNumMissed Declared in FinalExam
setScore Inherited from GradedActivity
getScore Inherited from GradedActivity
getGrade Inherited from GradedActivity
```

Notice the `GradedActivity` class's `score` field is not listed among the members of the `FinalExam` class. That is because the `score` field is private. Private members of the superclass cannot be accessed by the subclass, so technically speaking, they are not inherited. When an object of the subclass is

created, the private members of the superclass exist in memory, but only methods in the superclass can access them. They are truly private to the superclass.

You will also notice that the superclass's constructor is not listed among the members of the `FinalExam` class. It makes sense that superclass constructors are not inherited because their purpose is to construct objects of the superclass. In the next section, we will discuss in more detail how superclass constructors operate.

To see how inheritance works in this example, let's take a closer look at the `FinalExam` constructor. The constructor accepts two arguments: the number of test questions on the exam, and the number of questions missed by the student. These values are assigned to the `numQuestions` and `numMissed` fields in lines 22 and 23. Then, the number of points for each question is calculated in line 27, and the numeric test score is calculated in line 28. The last statement in the constructor, in line 32, reads:

```
setScore(numericScore);
```

This is a call to the `setScore` method. Although no `setScore` method appears in the `FinalExam` class, the method is inherited from the `GradedActivity` class. The program in [Code Listing 9-4](#) demonstrates the `FinalExam` class.

## Code Listing 9-4 (`FinalExamDemo.java`)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the FinalExam class, which
5 * inherits from the GradedActivity class.
6 */
7
8 public class FinalExamDemo
9 {
10 public static void main(String[] args)
11 {
```

```

12 int questions, // Number of questions
13 missed; // Number of questions missed
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the number of questions on the final exam.
19 System.out.print("How many questions are on " +
20 "the final exam? ");
21 questions = keyboard.nextInt();
22
23 // Get the number of questions the student missed.
24 System.out.print("How many questions did the " +
25 "student miss? ");
26 missed = keyboard.nextInt();
27
28 // Create a FinalExam object.
29 FinalExam exam = new FinalExam(questions, missed);
30
31 // Display the test results.
32 System.out.println("Each question counts " +
33 exam.getPointsEach() +
34 " points.");
35 System.out.println("The exam score is " +
36 exam.getScore());
37 System.out.println("The exam grade is " +
38 exam.getGrade());
39 }
40 }
```

## Program Output with Example Input Shown in Bold

How many questions are on the final exam? **20** 

How many questions did the student miss? **3** 

Each question counts 5.0 points.

The exam score is 85.0

The exam grade is B

In line 29, this program creates an instance of the `FinalExam` class and assigns its address to the `exam` variable. When a `FinalExam` object is created in memory, it has not only the members declared in the `FinalExam` class, but the members declared in the `GradedActivity` class as well. Notice, in lines 36 and 38, two public methods of the `GradedActivity` class, `getScore` and

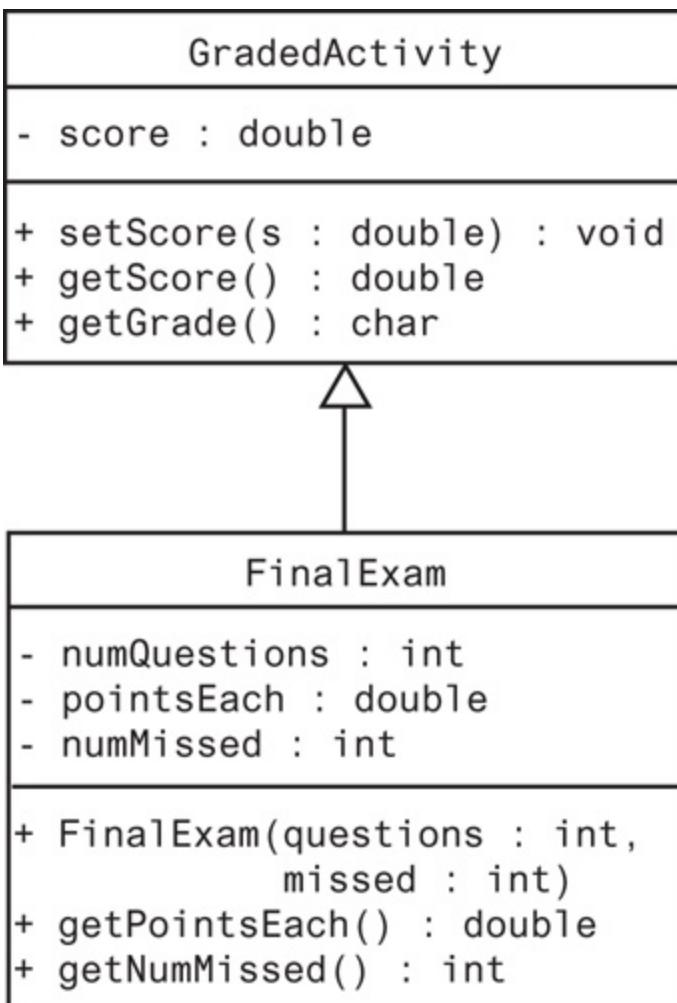
`getGrade`, are directly called using the object referenced by `exam`. When a class inherits from another class, the public members of the superclass become public members of the subclass. In this program, the `getScore` and `getGrade` methods can be called from the `exam` object because they are public members of the object's superclass.

As mentioned before, the private members of the superclass (in this case, the `score` field) cannot be accessed by the subclass. When the `exam` object is created in memory, a `score` field exists, but only the methods defined in the superclass, `GradedActivity`, can access it. It is truly private to the superclass. Because the `FinalExam` constructor cannot directly access the `score` field, it must call the superclass's `setScore` method (which is public) to store a value in it.

## Inheritance in UML Diagrams

You show inheritance in a UML diagram by connecting two classes with a line that has an open arrowhead at one end. The arrowhead points to the superclass. [Figure 9-5](#) shows a UML diagram depicting the relationship between the `GradedActivity` and `FinalExam` classes.

### Figure 9-5 UML diagram showing inheritance



[Figure 9-5 Full Alternative Text](#)

## The Superclass's Constructor

As was mentioned earlier, the `GradedActivity` class has only one constructor, which is the default constructor that Java automatically generated for it. When a `FinalExam` object is created, the `GradedActivity` class's default constructor is executed just before the `FinalExam` constructor is executed. In an inheritance relationship, the superclass constructor always executes before the subclass constructor.

[Code Listing 9-5](#) shows a class, `SuperClass1`, that has a programmer-defined no-arg constructor. The constructor simply displays the message “This is the

superclass constructor.” [Code Listing 9-6](#) shows SubClass1, which inherits from SuperClass1. This class also has a programmer-defined no-arg constructor, which displays the message “This is the subclass constructor.”

## Code Listing 9-5 (SuperClass1.java)

```
1 public class SuperClass1
2 {
3 // Constructor
4 public SuperClass1()
5 {
6 System.out.println("This is the superclass " +
7 "constructor.");
8 }
9 }
```

## Code Listing 9-6 (SubClass1.java)

```
1 public class SubClass1 extends SuperClass1
2 {
3 // Constructor
4 public SubClass1()
5 {
6 System.out.println("This is the subclass " +
7 "constructor.");
8 }
9 }
```

The program in [Code Listing 9-7](#) creates a SubClass1 object. As you can see from the program output, the superclass constructor executes first, followed by the subclass constructor.

## Code Listing 9-7

## (ConstructorDemo1.java)

```
1 /**
2 * This program demonstrates the order in which superclass
3 * and subclass constructors are called.
4 */
5
6 public class ConstructorDemo1
7 {
8 public static void main(String[] args)
9 {
10 SubClass1 obj = new SubClass1();
11 }
12 }
```

### Program Output

This is the superclass constructor.  
This is the subclass constructor.

## Inheritance Does Not Work in Reverse

In an inheritance relationship, the subclass inherits members from the superclass, not the other way around. This means it is not possible for a superclass to call a subclass's method. For example, if we create a `GradedActivity` object, it cannot call the `getPointsEach` or the `getNumMissed` methods, because they are members of the `FinalExam` class.



## Checkpoint

- 9.1 Here is the first line of a class declaration. What is the name of the superclass? What is the name of the subclass?

public class Truck extends Vehicle

2. 9.2 Look at the following class declarations, and answer the questions that follow them.

```
public class Shape
{
 private double area;

 public void setArea(double a)
 {
 area = a;
 }
 public double getArea()
 {
 return area;
 }
}
public class Circle extends Shape
{
 private double radius;

 public void setRadius(double r)
 {
 radius = r;
 setArea(Math.PI * r * r);
 }

 public double getRadius()
 {
 return radius;
 }
}
```

1. Which class is the superclass? Which class is the subclass?
2. Draw a UML diagram showing the relationship between these two classes.
3. When a `Circle` object is created, what are its public members?
4. What members of the `Shape` class are not accessible to the `Circle` class's methods?
5. Assume a program has the following declarations:

```
Shape s = new Shape();
Circle c = new Circle();
```

Indicate whether the following statements are legal or illegal:

```
c.setRadius(10.0);
s.setRadius(10.0);
System.out.println(c.getArea());
System.out.println(s.getArea());
```

3. 9.3 Class B inherits from class A. Describe the order in which the class's constructors execute when a class B object is created.

# 9.2 Calling the Superclass Constructor

## Concept:

The super key word refers to an object's superclass. You can use the super key word to call a superclass constructor.

In the previous section, you learned that a superclass's default constructor or no-arg constructor is automatically called just before the subclass's constructor executes. But what if the superclass does not have a default constructor or a no-arg constructor? Or, what if the superclass has multiple overloaded constructors, and you want to make sure a specific one is called? In either of these situations, you use the super key word to explicitly call a superclass constructor. The super key word refers to an object's superclass, and can be used to access members of the superclass.

[Code Listing 9-8](#) shows a class, SuperClass2, which has a no-arg constructor and a constructor that accepts an int argument. [Code Listing 9-9](#) shows SubClass2, which inherits from SuperClass2. This class's constructor uses the super key word to call the superclass's constructor and pass an argument to it.

## Code Listing 9-8 (SuperClass2.java)

```
1 public class SuperClass2
2 {
3 // No-arg constructor
4 public SuperClass2()
```

```

5 {
6 System.out.println("This is the superclass " +
7 "no-arg constructor.");
8 }
9
10 // Constructor #2
11 public SuperClass2(int arg)
12 {
13 System.out.println("The following argument was " +
14 "passed to the superclass " +
15 "constructor: " + arg);
16 }
17 }
```

## Code Listing 9-9 (SubClass2.java)

```

1 public class SubClass2 extends SuperClass2
2 {
3 // Constructor
4 public SubClass2()
5 {
6 // Call the superclass constructor.
7 super(10);
8
9 // Display a message.
10 System.out.println("This is the subclass " +
11 "constructor.");
12 }
13 }
```

In the SubClass2 constructor, the statement in line 7 calls the superclass constructor and passes the argument 10 to it. Here are three guidelines you should remember about calling a superclass constructor:

- The super statement that calls the superclass constructor can be written only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- The super statement that calls the superclass constructor must be the first statement in the subclass's constructor. This is because the

superclass's constructor must execute before the code in the subclass's constructor executes.

- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call the superclass's default constructor, or no-arg constructor, just before the code in the subclass's constructor executes. This is equivalent to placing the following statement at the beginning of a subclass constructor:

```
super();
```

The program in [Code Listing 9-10](#) demonstrates these classes.

## Code Listing 9-10 (ConstructorDemo2.java)

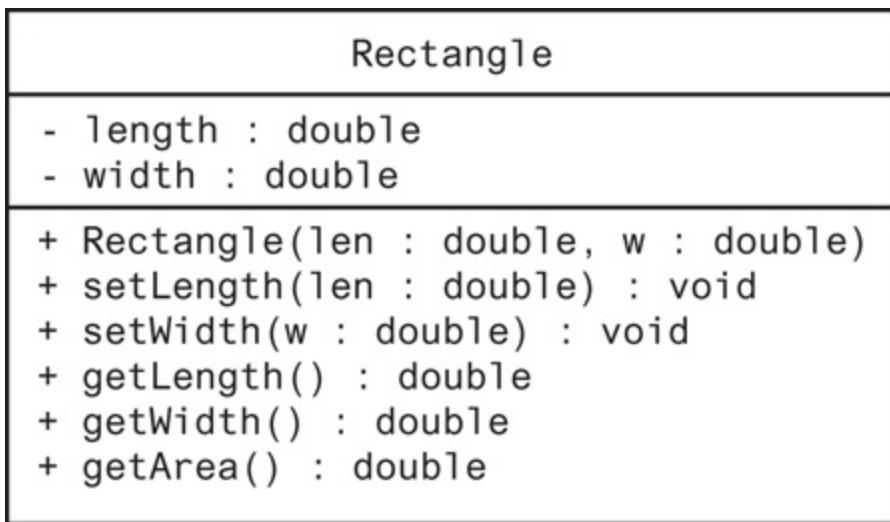
```
1 /**
2 * This program demonstrates how a superclass constructor
3 * can be called with the super key word.
4 */
5
6 public class ConstructorDemo2
7 {
8 public static void main(String[] args)
9 {
10 SubClass2 obj = new SubClass2();
11 }
12 }
```

### Program Output

The following argument was passed to the superclass constructor:  
This is the subclass constructor.

Let's look at a more meaningful example. Recall the Rectangle class that was introduced in [Chapter 3](#). [Figure 9-6](#) shows a UML diagram for the class.

# Figure 9-6 UML diagram for the Rectangle class



[Figure 9-6 Full Alternative Text](#)

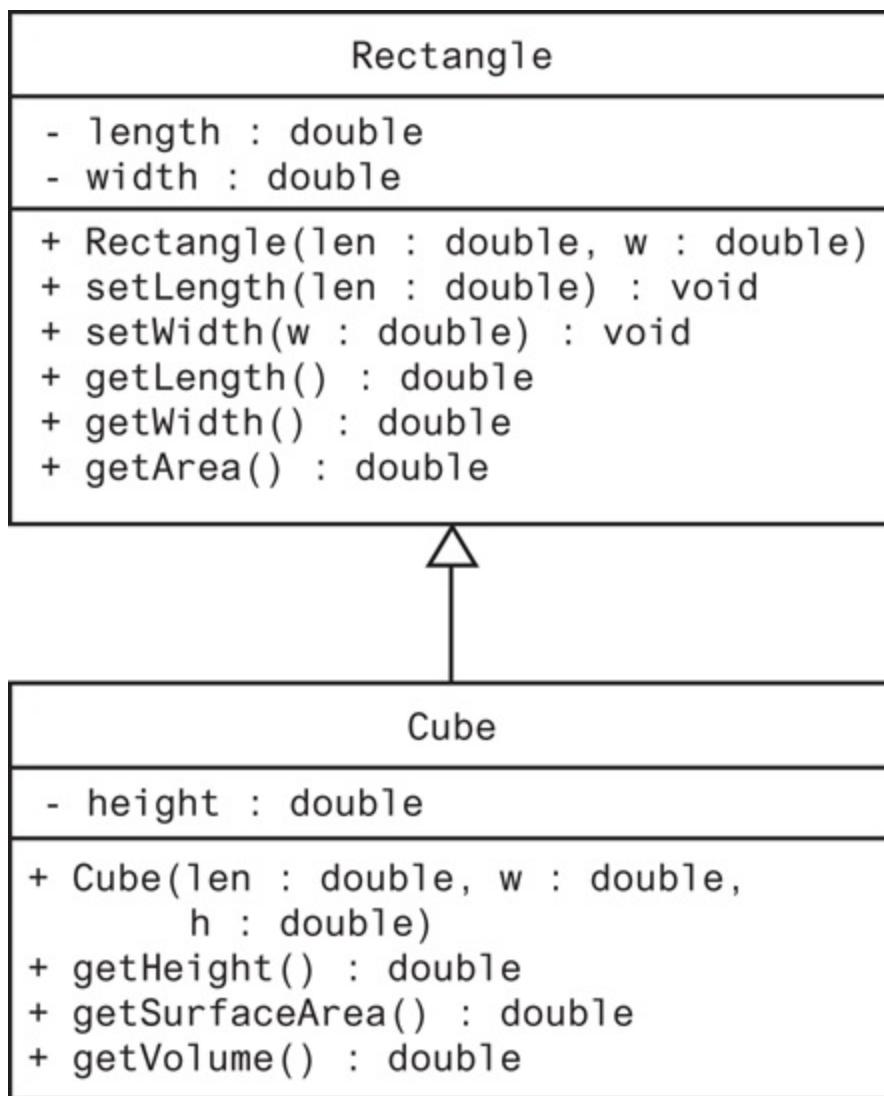
Here is part of the class's code:

```
public class Rectangle
{
 private double length;
 private double width;
 /**
 * Constructor
 */
 public Rectangle(double len, double w)
 {
 length = len;
 width = w;
 }
 (Other methods follow . . .)
}
```

Next we will design a `Cube` class, which inherits from the `Rectangle` class. The `Cube` class is designed to hold data about cubes, which not only have a length, width, and area (the area of the base), but a height, surface area, and volume as well. A UML diagram showing the inheritance relationship

between the Cube and Rectangle classes is shown in [Figure 9-7](#), and the code for the Cube class is shown in [Code Listing 9-11](#).

## Figure 9-7 UML diagram for the Rectangle and Cube classes



[Figure 9-7 Full Alternative Text](#)

## Code Listing 9-11 (Cube.java)

```
1 /**
2 * This class holds data about a cube.
3 */
4
5 public class Cube extends Rectangle
6 {
7 private double height; // The height of the cube
8
9 /**
10 * The constructor accepts the cube's length,
11 * width, and height as arguments.
12 */
13
14 public Cube(double len, double w, double h)
15 {
16 // Call the superclass constructor to
17 // initialize length and width.
18 super(len, w);
19
20 // Initialize height.
21 height = h;
22 }
23
24 /**
25 * The getHeight method returns the height
26 * field.
27 */
28
29 public double getHeight()
30 {
31 return height;
32 }
33
34 /**
35 * The getSurfaceArea method returns the
36 * cube's surface area.
37 */
38
39 public double getSurfaceArea()
40 {
41 return getArea() * 6;
42 }
43
44 /**
45 * The getVolume method returns the volume of
46 * the cube.
47 */
48
```

```
49 public double getVolume()
50 {
51 return getArea() * height;
52 }
53 }
```

The Cube constructor accepts arguments for the parameters `w`, `len`, and `h`. The values that are passed to `w` and `len` are subsequently passed as arguments to the Rectangle constructor in line 18. When the Rectangle constructor finishes, the remaining code in the Cube constructor is executed. The program in [Code Listing 9-12](#) demonstrates the class.

## Code Listing 9-12 (CubeDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates passing arguments to a
5 * superclass constructor.
6 */
7
8 public class CubeDemo
9 {
10 public static void main(String[] args)
11 {
12 double length, // To hold a length
13 width, // To hold a width
14 height; // To hold a height
15
16 // Create a Scanner object for keyboard input.
17 Scanner keyboard = new Scanner(System.in);
18
19 // Get the dimensions of a cube from the user.
20 System.out.println("Enter the following dimensions " +
21 "of a cube: ");
22 System.out.print("Length: ");
23 length = keyboard.nextDouble();
24 System.out.print("Width: ");
25 width = keyboard.nextDouble();
26 System.out.print("Height: ");
27 height = keyboard.nextDouble();
28 }
```

```

29 // Create a cube object and pass the dimensions
30 // to the constructor.
31 Cube myCube = new Cube(length, width, height);
32
33 // Display the properties of the cube.
34 System.out.println();
35 System.out.println("Here are the properties of " +
36 "the cube.");
37 System.out.println("Length: " + myCube.getLength());
38 System.out.println("Width: " + myCube.getWidth());
39 System.out.println("Height: " + myCube.getHeight());
40 System.out.println("Base Area: " + myCube.getArea());
41 System.out.println("Surface Area: " +
42 myCube.getSurfaceArea());
43 System.out.println("Volume: " + myCube.getVolume());
44 }
45 }

```

## Program Output with Example Input Shown in Bold

Enter the following dimensions of a cube:

Length: **10** 

Width: **15** 

Height: **12** 

Here are the properties of the cube.

Length: 10.0

Width: 15.0

Height: 12.0

Base Area: 150.0

Surface Area: 900.0

Volume: 1800.0

# When the Superclass Has No Default or No-Arg Constructor

Recall from [Chapter 3](#) that Java provides a default constructor for a class only when you provide no constructors for the class. This makes it possible to have a class with no default constructor. The Rectangle class we just looked at is an example. It has a constructor that accepts two arguments. Because we

have provided this constructor, the `Rectangle` class does not have a default constructor. In addition, we have not written a no-arg constructor for the class.

If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it *must* call one of the constructors that the superclass does have. If it does not, an error will result when the subclass is compiled.

## Summary of Constructor Issues in Inheritance

We have covered a number of important issues that you should remember about constructors in an inheritance relationship. The following list summarizes them:

- The superclass constructor always executes before the subclass constructor.
- You can write a `super` statement that calls a superclass constructor, but only in the subclass's constructor. You cannot call the superclass constructor from any other method.
- If a `super` statement that calls a superclass constructor appears in a subclass constructor, it must be the first statement.
- If a subclass constructor does not explicitly call a superclass constructor, Java will automatically call `super()` just before the code in the subclass's constructor executes.
- If a superclass does not have a default constructor and does not have a no-arg constructor, then a class that inherits from it *must* call one of the constructors that the superclass does have.



# Checkpoint

1. 9.4 Look at the following classes:

```
public class Ground
{
 public Ground()
 {
 System.out.println("You are on the ground.");
 }
}
public class Sky extends Ground
{
 public Sky()
 {
 System.out.println("You are in the sky.");
 }
}
```

What will the following program display?

```
public class Checkpoint
{
 public static void main(String[] args)
 {
 Sky object = new Sky();
 }
}
```

2. 9.5 Look at the following classes:

```
public class Ground
{
 public Ground()
 {
 System.out.println("You are on the ground.");
 }
 public Ground(String groundColor)
 {
 System.out.println("The ground is " + groundColor);
 }
}
public class Sky extends Ground
```

```
{
 public Sky()
 {
 System.out.println("You are in the sky.");
 }
 public Sky(String skyColor)
 {
 super("green");
 System.out.println("The sky is " + skyColor);
 }
}
```

What will the following program display?

```
public class Checkpoint
{
 public static void main(String[] args)
 {
 Sky object = new Sky("blue");
 }
}
```

# 9.3 Overriding Superclass Methods

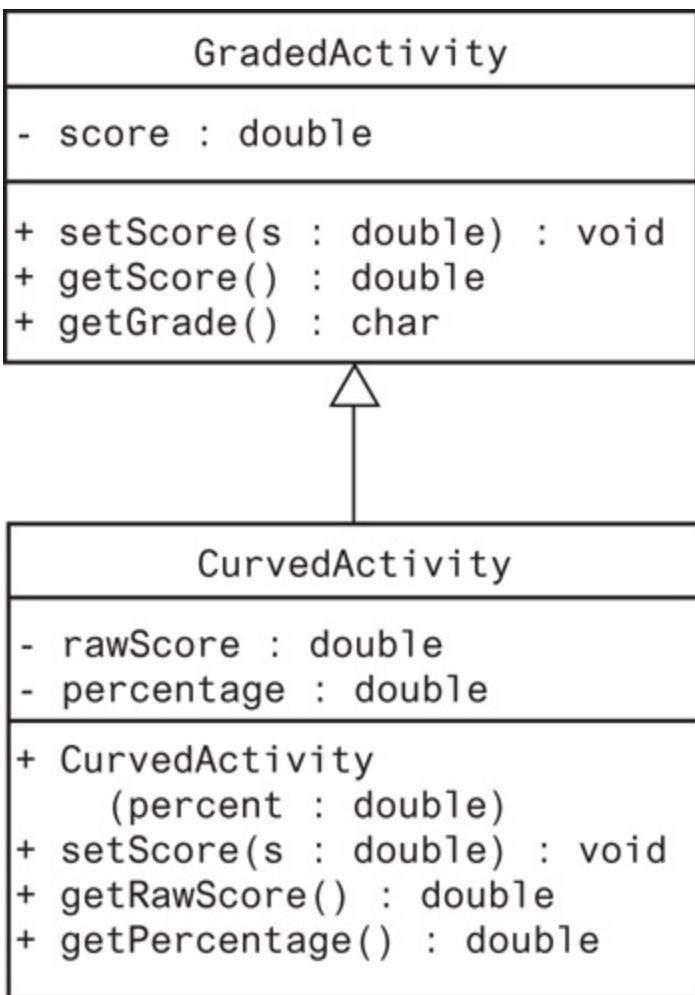
## Concept:

A subclass may have a method with the same signature as a superclass method. In such a case, the subclass method overrides the superclass method.

Sometimes a subclass inherits a method from its superclass, but the method is inadequate for the subclass's purpose. Because the subclass is more specialized than the superclass, it is sometimes necessary for the subclass to replace inadequate superclass methods with more suitable ones. This is known as *method overriding*.

For example, recall the `GradedActivity` class presented earlier in this chapter. This class has a `setScore` method that sets a numeric score, and a `getGrade` method that returns a letter grade based on that score. But, suppose a teacher wants to curve a numeric score before the letter grade is determined. For example, Dr. Harrison determines that in order to curve the grades in her class, she must multiply each student's score by a certain percentage. This gives an adjusted score that is used to determine the letter grade. To satisfy this need, we can design a new class, `CurvedActivity`, which inherits from the `GradedActivity` class and has its own specialized version of the `setScore` method. The `setScore` method in the subclass *overrides* the `setScore` method in the superclass. [Figure 9-8](#) shows a UML diagram depicting the relationship between the `GradedActivity` class and the `CurvedActivity` class.

## Figure 9-8 The GradedActivity and CurvedActivity classes



[Figure 9-8 Full Alternative Text](#)

[Table 9-1](#) summarizes the CurvedActivity class's fields, and [Table 9-2](#) summarizes the class's methods.

## Table 9-1 CurvedActivity class fields

| Field      | Description                                                                                          |
|------------|------------------------------------------------------------------------------------------------------|
| rawScore   | This field holds the student's unadjusted score.                                                     |
| percentage | This field holds the value by which the unadjusted score must be multiplied to get the curved score. |

# Table 9-2 CurvedActivity class methods

| Method        | Description                                                                                                                                                                                                                                                                                 |
|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constructor   | The constructor accepts a double argument that is the curve percentage. This value is assigned to the percentage field and the rawScore field is assigned 0.0.                                                                                                                              |
| setScore      | This method overrides the setScore method in the superclass. It accepts a double argument that is the student's unadjusted score. The method stores the argument in the rawScore field, then passes the result of rawScore * percentage as an argument to the superclass's setScore method. |
| getRawScore   | This method returns the value in the rawScore field.                                                                                                                                                                                                                                        |
| getPercentage | This method returns the value in the percentage field.                                                                                                                                                                                                                                      |

[Code Listing 9-13](#) shows the CurvedActivity class. The setScore method appears in lines 31 through 35. It is important to note that the setScore method in the CurvedActivity class has the same signature as the setScore method in the superclass GradedActivity. In order for overriding to occur, the subclass method must have the same signature as the superclass method. When an object of the subclass invokes the setScore method, it invokes the subclass's version of the method, not the superclass's.



## Note:

Recall from [Chapter 6](#), a method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear.

# Code Listing 9-13

## (CurvedActivity.java)

```
1 /**
2 * This class computes a curved grade. It extends
3 * the GradedActivity class.
4 */
5
6 public class CurvedActivity extends GradedActivity
7 {
8 double rawScore; // Unadjusted score
9 double percentage; // Curve percentage
10
11 /**
12 * The constructor sets the curve percentage.
13 */
14
15 public CurvedActivity(double percent)
16 {
17 percentage = percent;
18 rawScore = 0.0;
19 }
20
21 /**
22 * The setScore method overrides the superclass setScore method.
23 * This version accepts the unadjusted score as an argument.
24 * score is multiplied by the curve percentage and the result
25 * sent as an argument to the superclass's setScore method.
26 */
27
28 @Override
29 public void setScore(double s)
30 {
31 rawScore = s;
32 super.setScore(rawScore * percentage);
33 }
34
35 /**
36 * The getRawScore method returns the raw score.
37 */
38
39 public double getRawScore()
40 {
41 return rawScore;
```

```
42 }
43
44 /**
45 * The getPercentage method returns the curve
46 * percentage.
47 */
48
49 public double getPercentage()
50 {
51 return percentage;
52 }
53 }
```

Notice in line 28, the `@Override` annotation appears just before the `setScore` method definition. This annotation tells the Java compiler that the `setScore` method is meant to override a method in the superclass.

The `@Override` annotation in line 28 is not required, but it is recommended that you use it. If the method fails to correctly override a method in the superclass, the compiler will display an error message. For example, suppose we had written the method header in line 29 like this:

```
public void setscore(double s)
```

If you look closely at the method name, you will see that all the letters are written in lowercase. This does not match the method's name in the superclass, which is `setScore`. Without the `@Override` annotation, the code would still compile and execute, but we would not get the expected results because the method in the subclass would not override the method in the superclass. However, by using the `@Override` annotation in line 28, the compiler would generate an error letting us know that the subclass method does not override any method in the superclass.

Let's take a closer look at the `setScore` method in the `CurvedActivity` class. It accepts an argument, which is the student's unadjusted numeric score. This value is stored in the `rawScore` field. Then the following statement, in line 32, is executed:

```
super.setScore(rawScore * percentage);
```

As you already know, the super key word refers to the object's superclass. This statement calls the superclass's version of the setScore method with the result of the expression rawScore \* percentage passed as an argument. This is necessary because the superclass's score field is private, and the subclass cannot access it directly. In order to store a value in the superclass's score field, the subclass must call the superclass's setScore method. A subclass may call an overridden superclass method by prefixing its name with the super key word and a dot (.). The program in [Code Listing 9-14](#) demonstrates this class.

## Code Listing 9-14 (CurvedActivityDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the CurvedActivity class,
5 * which inherits from the GradedActivity class.
6 */
7
8 public class CurvedActivityDemo
9 {
10 public static void main(String[] args)
11 {
12 double score, // Raw score
13 curvePercent; // Curve percentage
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the unadjusted exam score.
19 System.out.print("Enter the student's raw " +
20 "numeric score: ");
21 score = keyboard.nextDouble();
22
23 // Get the curve percentage.
24 System.out.print("Enter the curve percentage: ");
25 curvePercent = keyboard.nextDouble();
26
27 // Create a CurvedActivity object.
28 CurvedActivity curvedExam =
```

```
29 new CurvedActivity(curvePercent);
30
31 // Set the exam score.
32 curvedExam.setScore(score);
33
34 // Display the test results.
35 System.out.println("The raw score is " +
36 curvedExam.getRawScore() +
37 " points.");
38 System.out.println("The curved score is " +
39 curvedExam.getScore());
40 System.out.println("The exam grade is " +
41 curvedExam.getGrade());
42 }
43 }
```

## Program Output with Example Input Shown in Bold

```
Enter the student's raw numeric score: 87 
Enter the curve percentage: 1.06 
The raw score is 87.0 points.
The curved score is 92.22
The exam grade is A
```

This program uses the `curvedExam` variable to reference a `CurvedActivity` object. The statement in line 32 calls the `setScore` method. Because `curvedExam` references a `CurvedActivity` object, this statement calls the `CurvedActivity` class's `setScore` method, not the superclass's version.

Even though a subclass may override a method in the superclass, superclass objects still call the superclass version of the method. For example, the following code creates an object of the `GradedActivity` class, and calls the `setScore` method:

```
GradedActivity regularExam = new GradedActivity();
regularExam.setScore(85);
```

Because `regularExam` references a `GradedActivity` object, this code calls the `GradedActivity` class's version of the `setScore` method.

# Overloading versus Overriding

There is a distinction between overloading a method and overriding a method. Recall from [Chapter 6](#), overloading is when a method has the same name as one or more other methods, but a different parameter list. Although overloaded methods have the same name, they have different signatures. When a method overrides another method, however, they both have the same signature.

Both overloading and overriding can take place in an inheritance relationship. You already know overloaded methods can appear within the same class. In addition, a method in a subclass can overload a method in the superclass. If class A is the superclass and class B is the subclass, a method in class B can overload a method in class A, or another method in class B. Overriding, on the other hand, can take place only in an inheritance relationship. If class A is the superclass and class B is the subclass, a method in class B can override a method in class A. However, a method cannot override another method in the same class. The following list summarizes the distinction between overloading and overriding:

- If two methods have the same name but different signatures, they are overloaded. This is true where the methods are in the same class, or where one method is in the superclass and the other method is in the subclass.
- If a method in a subclass has the same signature as a method in the superclass, the subclass method overrides the superclass method.

The distinction between overloading and overriding is important because it can affect the accessibility of superclass methods in a subclass. When a subclass overloads a superclass method, both methods can be called with a subclass object. However, when a subclass overrides a superclass method, only the subclass's version of the method can be called with a subclass object. For example, look at the `SuperClass3` class in [Code Listing 9-15](#). It has two overloaded methods named `showValue`. One of the methods accepts an `int` argument, and the other accepts a `String` argument.

## Code Listing 9-15 (SuperClass3.java)

```
1 public class SuperClass3
2 {
3 /**
4 * The following method displays an int.
5 */
6
7 public void showValue(int arg)
8 {
9 System.out.println("SUPERCLASS: The int argument was " + ar
10 }
11
12 /**
13 * The following method displays a String.
14 */
15
16 public void showValue(String arg)
17 {
18 System.out.println("SUPERCLASS: The String argument was " +
19 }
20 }
```

Now look at the SubClass3 class in [Code Listing 9-16](#). It inherits from the SuperClass3 class.

## Code Listing 9-16 (SubClass3.java)

```
1 public class SubClass3 extends SuperClass3
2 {
3 /**
4 * This method overrides one of the superclass methods.
5 */
6
7 @Override
8 public void showValue(int arg)
9 {
10 System.out.println("SUBCLASS: The int argument was " + arg)
11 }
```

```

12 /**
13 * This method overloads the superclass methods.
14 */
15
16
17 public void showValue(double arg)
18 {
19 System.out.println("SUBCLASS: The double argument was " + a
20 }
21 }
```

Notice SubClass3 also has two methods named `showValue`. The first one, in lines 8 through 11, accepts an `int` argument. This method overrides one of the superclass methods because they have the same signature. The second `showValue` method, in lines 17 through 20, accepts a `double` argument. This method overloads the other `showValue` methods because none of the others have the same signature. Although there is a total of four `showValue` methods in these classes, only three of them can be called from a `SubClass3` object. This is demonstrated in [Code Listing 9-17](#).

## Code Listing 9-17 (ShowValueDemo.java)

```

1 /**
2 * This program demonstrates the methods in the
3 * SuperClass3 and SubClass3 classes.
4 */
5
6 public class ShowValueDemo
7 {
8 public static void main(String[] args)
9 {
10 SubClass3 myObject = new SubClass3();
11
12 myObject.showValue(10); // Pass an int.
13 myObject.showValue(1.2); // Pass a double.
14 myObject.showValue("Hello"); // Pass a String.
15 }
16 }
```

## **Program Output**

```
SUBCLASS: The int argument was 10
SUBCLASS: The double argument was 1.2
SUPERCLASS: The String argument was Hello
```

When an `int` argument is passed to `showValue`, the subclass's method is called because it overrides the superclass method. In order to call the overridden superclass method, we would have to use the `super` key word in the subclass method. Here is an example:

```
public void showValue(int arg)
{
 super.showValue(arg); // Call the superclass method.
 System.out.println("SUBCLASS: The int argument was " + arg);
}
```

## **Preventing a Method from Being Overridden**

When a method is declared with the `final` modifier, it cannot be overridden in a subclass. The following method header is an example that uses the `final` modifier:

```
public final void message()
```

If a subclass attempts to override a `final` method, the compiler generates an error. This technique can be used to make sure a particular superclass method is used by subclasses, and not a modified version of it.



## **Checkpoint**

1. 9.6 Under what circumstances would a subclass need to override a superclass method?

2. 9.7 How can a subclass method call an overridden superclass method?
3. 9.8 If a method in a subclass has the same signature as a method in the superclass, does the subclass method overload or override the superclass method?
4. 9.9 If a method in a subclass has the same name as a method in the superclass, but uses a different parameter list, does the subclass method overload or override the superclass method?
5. 9.10 How do you prevent a method from being overridden?

# **9.4 Protected Members**

## **Concept:**

Protected members of a class can be accessed by methods in a subclass, and by methods in the same package as the class.

Until now, you have used two access specifications within a class: `private` and `public`. Java provides a third access specification, `protected`. A protected member of a class can be directly accessed by methods of the same class or methods of a subclass. In addition, protected members can be accessed by methods of any class that are in the same package as the protected member's class. A protected member is not quite private, because it can be accessed by some methods outside the class. Protected members are not quite public either because access to them is restricted to methods in the same class, subclasses, and classes in the same package as the member's class. A protected member's access is somewhere between private and public.

Let's look at a class with a protected member. [Code Listing 9-18](#) shows the `GradedActivity2` class, which is a modification of the `GradedActivity` class presented earlier. In this class, the `score` field has been made protected instead of private.

## **Code Listing 9-18 (`GradedActivity2.java`)**

```
1 /**
2 * A class that holds a grade for a graded activity.
3 */
4
5 public class GradedActivity2
```

```
6 {
7 protected double score; // Numeric score
8
9 /**
10 * The setScore method stores its argument in
11 * the score field.
12 */
13
14 public void setScore(double s)
15 {
16 score = s;
17 }
18
19 /**
20 * The getScore method returns the score field.
21 */
22
23 public double getScore()
24 {
25 return score;
26 }
27
28 /**
29 * The getGrade method returns a letter grade
30 * determined from the score field.
31 */
32
33 public char getGrade()
34 {
35 char letterGrade; // To hold the grade
36
37 if (score >= 90)
38 letterGrade = 'A';
39 else if (score >= 80)
40 letterGrade = 'B';
41 else if (score >= 70)
42 letterGrade = 'C';
43 else if (score >= 60)
44 letterGrade = 'D';
45 else
46 letterGrade = 'F';
47
48 return letterGrade;
49 }
50 }
```

Because the score field is declared as protected, any class that inherits from this class has direct access to it. The FinalExam2 class, shown in [Code Listing 9-19](#), is an example. This class is a modification of the FinalExam class, which was presented earlier. This class has a new method, adjustScore, which directly accesses the superclass's score field. If the contents of score have a fractional part of .5 or greater, the method rounds score up to the next whole number. The adjustScore method is called from the constructor.

## Code Listing 9-19 (FinalExam2.java)

```
1 /**
2 * This class determines the grade for a final exam. The
3 * numeric score is rounded up to the next whole number
4 * if its fractional part is .5 or greater.
5 */
6
7 public class FinalExam2 extends GradedActivity2
8 {
9 private int numQuestions; // Number of questions
10 private double pointsEach; // Points for each question
11 private int numMissed; // Number of questions missed
12
13 /**
14 * The constructor accepts as arguments the number
15 * of questions on the exam and the number of
16 * questions the student missed.
17 */
18
19 public FinalExam2(int questions, int missed)
20 {
21 double numericScore; // To hold the numeric score
22
23 // Set the numQuestions and numMissed fields.
24 numQuestions = questions;
25 numMissed = missed;
26
27 // Calculate the points for each question and
28 // the numeric score for this exam.
29 pointsEach = 100.0 / questions;
30 numericScore = 100.0 - (missed * pointsEach);
```

```
31 // Call the superclass's setScore method to
32 // set the numeric score.
33 setScore(numericScore);
34 adjustScore();
35 }
36 /**
37 * The getPointsEach method returns the pointsEach
38 * field.
39 */
40
41 public double getPointsEach()
42 {
43 return pointsEach;
44 }
45
46 /**
47 * The getNumMissed method returns the numMissed
48 * field.
49 */
50
51 public int getNumMissed()
52 {
53 return numMissed;
54 }
55
56 /**
57 * The adjustScore method adjusts a numeric score.
58 * If score is within 0.5 points of the next whole
59 * number, it rounds the score up.
60 */
61
62 private void adjustScore()
63 {
64 double fraction; // Fractional part of a score
65
66 // Get the fractional part of the score.
67 fraction = score - (int) score;
68
69 // If the fractional part is 0.5 or greater,
70 // round the score up to the next whole number.
71 if (fraction >= 0.5)
72 score = score + (1.0 - fraction);
73 }
74
75 }
76 }
```

The program in [Code Listing 9-20](#) demonstrates the FinalExam2 class.

## Code Listing 9-20 (ProtectedDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the FinalExam2 class, which
5 * inherits from the GradedActivity2 class.
6 */
7
8 public class ProtectedDemo
9 {
10 public static void main(String[] args)
11 {
12 int questions, // Number of questions
13 missed; // Number of questions missed
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the number of questions on the final exam.
19 System.out.print("How many questions are on " +
20 "the final exam? ");
21 questions = keyboard.nextInt();
22
23 // Get the number of questions the student missed.
24 System.out.print("How many questions did the " +
25 "student miss? ");
26 missed = keyboard.nextInt();
27
28 // Create a FinalExam2 object.
29 FinalExam2 exam =
30 new FinalExam2(questions, missed);
31
32 // Display the test results.
33 System.out.println("Each question counts " +
34 exam.getPointsEach() +
35 " points.");
36 System.out.println("The exam score is " +
37 exam.getScore());
38 System.out.println("The exam grade is " +
```

```
39 exam.getGrade());
40 }
41 }
```

## Program Output with Example Input Shown in Bold

How many questions are on the final exam? **40**

How many questions did the student miss? **5**

Each question counts 2.5 points.

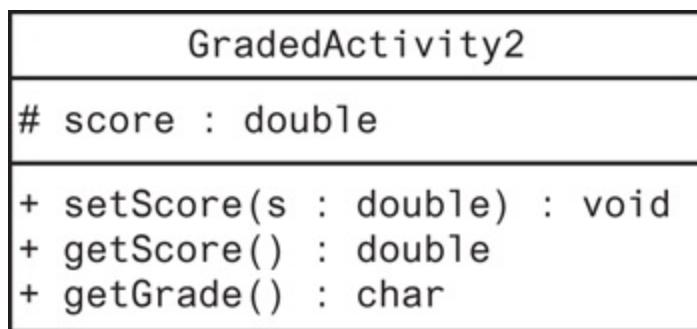
The exam score is 88.0

The exam grade is B

In the example running of the program, the student missed 5 out of 40 questions. The unadjusted numeric score would be 87.5, but the adjustScore method rounded the score field up to 88.

Protected class members can be denoted in a UML diagram with the # symbol. [Figure 9-9](#) shows a UML diagram for the GradedActivity2 class, with the score field denoted as protected.

## Figure 9-9 UML diagram for the GradedActivity2 class



[Figure 9-9 Full Alternative Text](#)

Although making a class member protected instead of private might make some tasks easier, you should avoid this practice when possible. This is

because any class that inherits from the class, or is in the same package, has unrestricted access to the protected member. It is always better to make all fields private, then provide public methods for accessing those fields.

## Package Access

If you do not provide an access specifier for a class member, the class member is given *package access* by default. This means that any method in the same package can access the member. Here is an example:

```
public class Circle
{
 double radius;
 int centerX, centerY;

 (Method definitions follow . . .)
}
```

In this class, the `radius`, `centerX`, and `centerY` fields were not given an access specifier, so the compiler grants them package access. Any method in the same package as the `Circle` class can directly access these members.

There is a subtle difference between protected access and package access. Protected members can be accessed by methods in the same package or in a subclass. This is true even if the subclass is in a different package. Members with package access, however, cannot be accessed by subclasses that are in a different package.

It is more likely that you will give package access to class members by accident than by design, because it is easy to forget the access specifier. Although there are circumstances under which package access can be helpful, you should normally avoid it. Be careful to always specify an access specifier for class members.

[Tables 9-3](#) and [9-4](#) summarize how each of the access specifiers affect a class member's accessibility within and outside of the class's package.

## Table 9-3 Accessibility from within the class's package

| Access Modifier | Accessible to a subclass inside the same package? | Accessible to all other classes in the same package? |
|-----------------|---------------------------------------------------|------------------------------------------------------|
| default         |                                                   |                                                      |
| (no modifier)   | Yes                                               | Yes                                                  |
| public          | Yes                                               | Yes                                                  |
| protected       | Yes                                               | Yes                                                  |
| private         | No                                                | No                                                   |

## Table 9-4 Accessibility from outside the class's package

| Access Modifier       | Accessible to a subclass outside the package? | Accessible to all other classes outside the package? |
|-----------------------|-----------------------------------------------|------------------------------------------------------|
| default (no modifier) | No                                            | No                                                   |
| public                | Yes                                           | Yes                                                  |
| protected             | Yes                                           | No                                                   |
| private               | No                                            | No                                                   |



## Checkpoint

1. 9.11 When a class member is declared as protected, what code can

access it?

2. 9.12 What is the difference between private members and protected members?
3. 9.13 Why should you avoid making class members protected when possible?
4. 9.14 What is the difference between private access and package access?
5. 9.15 Why is it easy to give package access to a class member by accident?

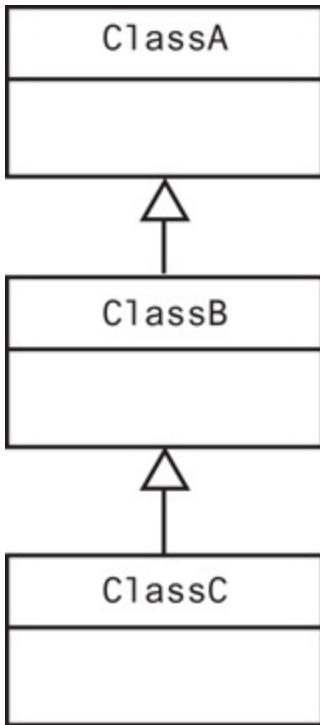
# **9.5 Classes That Inherit from Subclasses**

## **Concept:**

A superclass can also inherit from another class.

Sometimes it is desirable to establish a chain of inheritance in which one class inherits from a second class, which in turn inherits from a third class, as illustrated by [Figure 9-10](#). In some cases, this chaining of classes goes on for many layers.

## **Figure 9-10 A chain of inheritance**



In [Figure 9-10](#), `ClassC` inherits `ClassB`'s members, including the ones that `ClassB` inherited from `ClassA`. Let's look at an example of such a chain of inheritance. Consider the `PassFailActivity` class, shown in [Code Listing 9-21](#), which inherits from the `GradedActivity` class. The class is intended to determine a letter grade of "P" for passing, or "F" for failing.

## Code Listing 9-21 (`PassFailActivity.java`)

```

1 /**
2 * This class holds a numeric score and determines
3 * whether the score is passing or failing.
4 */
5
6 public class PassFailActivity extends GradedActivity
7 {
8 private double minPassingScore; // Minimum passing score
9
10 /**
11 * The constructor accepts the minimum passing
12 * score as its argument.

```

```
13 */
14
15 public PassFailActivity(double mps)
16 {
17 minPassingScore = mps;
18 }
19
20 /**
21 * The getGrade method returns a letter grade determined
22 * from the score field. This method overrides the getGrade
23 * method in the superclass.
24 */
25
26 @Override
27 public char getGrade()
28 {
29 char letterGrade; // To hold the letter grade
30
31 if (super.getScore() >= minPassingScore)
32 letterGrade = 'P';
33 else
34 letterGrade = 'F';
35
36 return letterGrade;
37 }
38 }
```

The `PassFailActivity` constructor, in lines 15 through 18, accepts a double argument that is the minimum passing grade for the activity. This value is stored in the `minPassingScore` field. The `getGrade` method in lines 27 through 37, which overrides the superclass method, returns a grade of "P" if the numeric score is greater than or equal to `minPassingScore`. Otherwise, the method returns a grade of "F".

Suppose we wish to extend this class with another class that is even more specialized. For example, the `PassFailExam` class, shown in [Code Listing 9-22](#), determines a passing or failing grade for an exam. It has fields for the number of questions on the exam (`numQuestions`), the number of points each question is worth (`pointsEach`), and the number of questions missed by the student (`numMissed`).

# Code Listing 9-22

## (PassFailExam.java)

```
1 /**
2 * This class determines a passing or failing grade for
3 * an exam.
4 */
5
6 public class PassFailExam extends PassFailActivity
7 {
8 private int numQuestions; // Number of questions
9 private double pointsEach; // Points for each question
10 private int numMissed; // Number of questions missed
11
12 /**
13 * The constructor accepts as arguments the number
14 * of questions on the exam, the number of
15 * questions the student missed, and the minimum
16 * passing score.
17 */
18
19 public PassFailExam(int questions, int missed,
20 double minPassing)
21 {
22 // Call the superclass constructor.
23 super(minPassing);
24
25 // Declare a local variable for the numeric score.
26 double numericScore;
27
28 // Set the numQuestions and numMissed fields.
29 numQuestions = questions;
30 numMissed = missed;
31
32 // Calculate the points for each question and
33 // the numeric score for this exam.
34 pointsEach = 100.0 / questions;
35 numericScore = 100.0 - (missed * pointsEach);
36
37 // Call the superclass's setScore method to
38 // set the numeric score.
39 setScore(numericScore);
40 }
41
```

```

42 /**
43 * The getPointsEach method returns the pointsEach
44 * field.
45 */
46
47 public double getPointsEach()
48 {
49 return pointsEach;
50 }
51
52 /**
53 * The getNumMissed method returns the numMissed
54 * field.
55 */
56
57 public int getNumMissed()
58 {
59 return numMissed;
60 }
61 }
```

The PassFailExam class inherits the PassFailActivity class's members, including the ones that PassFailActivity inherited from GradedActivity. The program in [Code Listing 9-23](#) demonstrates the class.

## Code Listing 9-23 (PassFailExamDemo.java)

```

1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the PassFailExam class.
5 */
6
7 public class PassFailExamDemo
8 {
9 public static void main(String[] args)
10 {
11 int questions, // Number of questions
12 missed; // Number of questions missed
13 double minPassing; // Minimum passing score
14 }
```

```

15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Get the number of questions on the exam.
19 System.out.print("How many questions are " +
20 "on the exam? ");
21 questions = keyboard.nextInt();
22
23 // Get the number of questions the student missed.
24 System.out.print("How many questions did the " +
25 "student miss? ");
26 missed = keyboard.nextInt();
27
28 // Get the minimum passing score.
29 System.out.print("What is the minimum " +
30 "passing score? ");
31 minPassing = keyboard.nextInt();
32
33 // Create a PassFailExam object.
34 PassFailExam exam =
35 new PassFailExam(questions, missed, minPassing);
36
37 // Display the test results.
38 System.out.println("Each question counts " +
39 exam.getPointsEach() +
40 " points.");
41 System.out.println("The exam score is " +
42 exam.getScore());
43 System.out.println("The exam grade is " +
44 exam.getGrade());
45 }
46 }

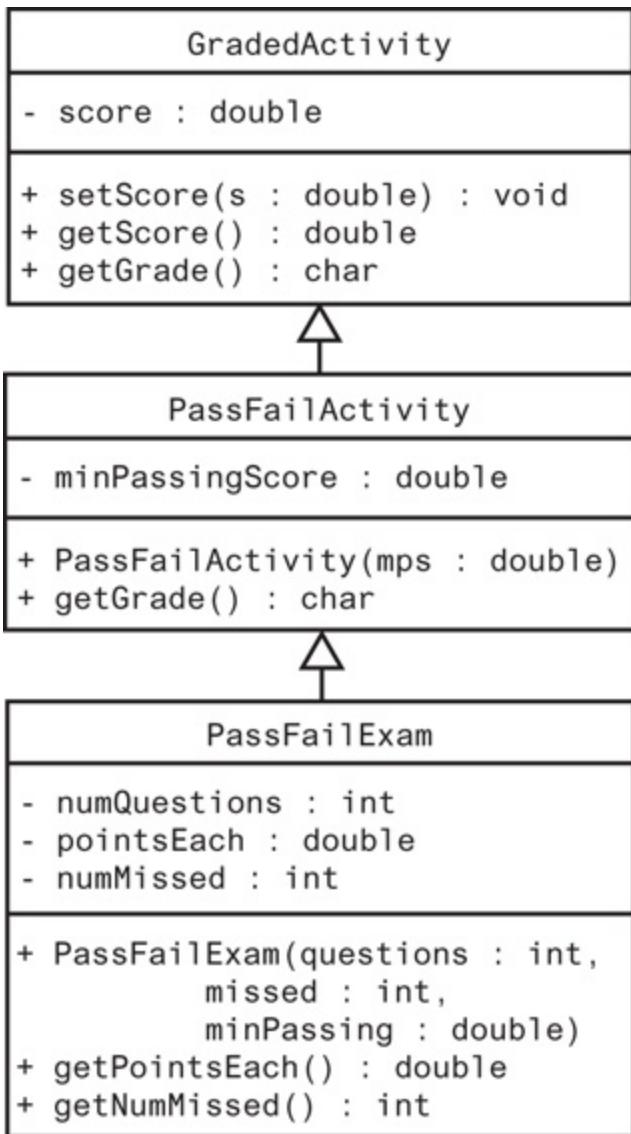
```

## Program Output with Example Input Shown in Bold

How many questions are on the exam? **100**   
 How many questions did the student miss? **25**   
 What is the minimum passing score? **60**   
 Each question counts 1.0 points.  
 The exam score is 75.0  
 The exam grade is P

[Figure 9-11](#) shows a UML diagram depicting the inheritance relationship among the GradedActivity, PassFailActivity, and PassFailExam classes.

# Figure 9-11 The GradedActivity, PassFailActivity, and PassFailExam classes

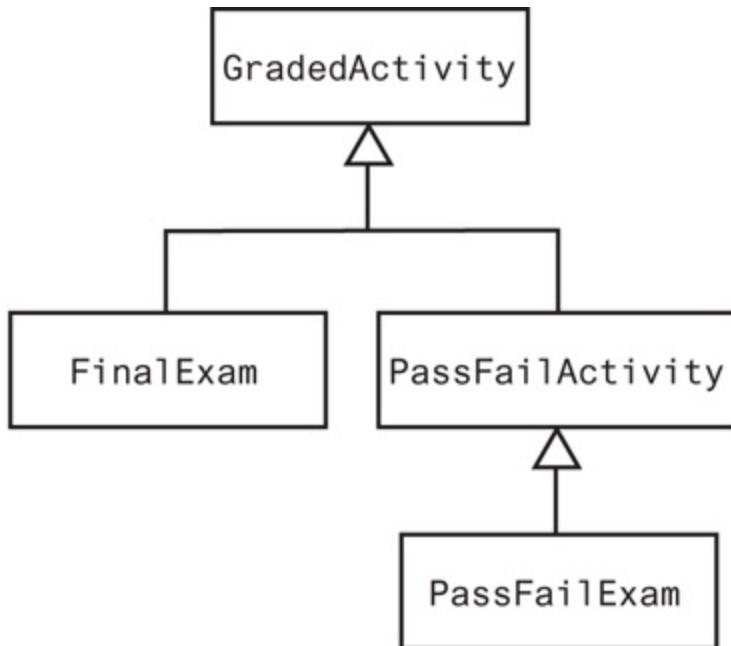


[Figure 9-11 Full Alternative Text](#)

# Class Hierarchies

Classes often are depicted graphically in a *class hierarchy*. Like a family tree, a class hierarchy shows the inheritance relationships among classes. [Figure 9-12](#) shows a class hierarchy for the `GradedActivity`, `FinalExam`, `PassFailActivity`, and `PassFailExam` classes. The more general classes are toward the top of the tree, and the more specialized classes are toward the bottom.

**Figure 9-12 Class hierarchy**



# 9.6 The Object Class

## Concept:

The Java API has a class named `Object`, which all other classes directly or indirectly inherit from.

Every class in Java, including the ones in the API and the classes that you create, directly or indirectly inherit from a class named `Object`, which is part of the `java.lang` package. Here's how it happens: When a class does not use the `extends` key word to inherit from another class, Java automatically extends it from the `Object` class. For example, look at the following class declaration:

```
public class MyClass
{
 (Member Declarations . . .)
}
```

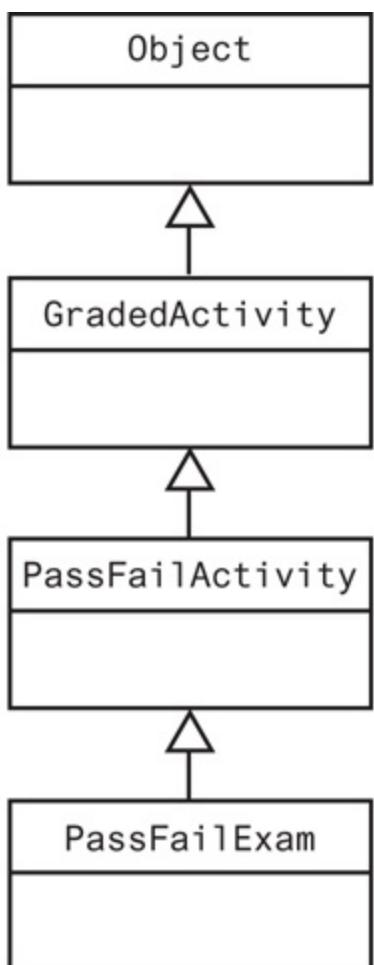
This class does not explicitly extend any other class, so Java treats it as though it was written as:

```
public class MyClass extends Object
{
 (Member Declarations . . .)
}
```

Ultimately, every class inherits from the `Object` class. [Figure 9-13](#) shows how the `PassFailExam` class ultimately inherits from `Object`.

**Figure 9-13 The line of inheritance from Object to**

# PassFailExam



Because every class directly or indirectly inherits from the `Object` class, every class inherits the `Object` class's members. Two of the most useful are the `toString` and `equals` methods. In [Chapter 6](#), you learned that every object has a `toString` and an `equals` method, and now you know why! It is because those methods are inherited from the `Object` class.

In the `Object` class, the `toString` method returns a string containing the object's class name, followed by the @ character, followed by the object's hexadecimal hashcode. (An object's hashcode is an integer that is unique to the object.) The `equals` method accepts the address of an object as its argument, and returns true if it is the same as the calling object's address. This is demonstrated in [Code Listing 9-24](#).

# Code Listing 9-24 (ObjectMethods.java)

```
1 /**
2 * This program demonstrates the toString and equals
3 * methods that are inherited from the Object class.
4 */
5
6 public class ObjectMethods
7 {
8 public static void main(String[] args)
9 {
10 // Create two objects.
11 PassFailExam exam1 = new PassFailExam(0, 0, 0);
12 PassFailExam exam2 = new PassFailExam(0, 0, 0);
13
14 // Send the objects to println, which will
15 // call the toString method.
16 System.out.println(exam1);
17 System.out.println(exam2);
18
19 // Test the equals method.
20 if (exam1.equals(exam2))
21 System.out.println("The two are the same.");
22 else
23 System.out.println("The two are not the same.");
24 }
25 }
```

## Program Output

```
PassFailExam@45a877
PassFailExam@1372a1a
The two are not the same.
```

If you wish to change the behavior of either of these methods for a given class, you must override them in the class.



## Checkpoint

1. 9.16 Look at the following class definition:

```
public class ClassD extends ClassB
{
 (Member Declarations . . .)
}
```

Because `ClassD` inherits from `ClassB`, is it true that `ClassD` does not inherit from the `Object` class? Why or why not?

2. 9.17 When you create a class, it automatically has a `toString` method and an `equals` method. Why?

# 9.7 Polymorphism

## Concept:

A reference variable can reference objects of classes that inherit from the variable's class.



**VideoNote** Polymorphism

Look at the following statement that declares a reference variable named exam:

```
GradedActivity exam;
```

This statement tells us that the exam variable's data type is GradedActivity. Therefore, we can use the exam variable to reference a GradedActivity object, as shown in the following statement:

```
exam = new GradedActivity();
```

The GradedActivity class is also used as the superclass for the FinalExam class. Because of the "is-a" relationship between a superclass and a subclass, an object of the FinalExam class is not just a FinalExam object. It is also a GradedActivity object. (A final exam *is a* graded activity.) Because of this relationship, we can use a GradedActivity variable to reference a FinalExam object. For example, look at the following statement:

```
GradedActivity exam = new FinalExam(50, 7);
```

This statement declares exam as a GradedActivity variable. It creates a FinalExam object and stores the object's address in the exam variable. This statement is perfectly legal, and will not cause an error message because a

`FinalExam` object is also a `GradedActivity` object.

This is an example of polymorphism. The term *polymorphism* means the ability to take many forms. In Java, a reference variable is *polymorphic* because it can reference objects of types different from its own, as long as those types are related to its type through inheritance. All of the following declarations are legal because the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes inherit from `GradedActivity`:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
```

Although a `GradedActivity` variable can reference objects of any class that inherits from `GradedActivity`, there is a limit to what the variable can do with those objects. Recall the `GradedActivity` class has three methods: `setScore`, `getScore`, and `getGrade`. So, a `GradedActivity` variable can be used to call only those three methods, regardless of the type of object the variable references. For example, look at the following code:

```
GradedActivity exam = new PassFailExam(100, 10, 70);
System.out.println(exam.getScore()); // This works.
System.out.println(exam.getGrade()); // This works.
System.out.println(exam.getPointsEach()); // ERROR! Won't work.
```

In this code, `exam` is declared as a `GradedActivity` variable and is assigned the address of a `PassFailExam` object. The `GradedActivity` class has only the `setScore`, `getScore`, and `getGrade` methods, so those are the only methods that the `exam` variable knows how to execute. The last statement in this code is a call to the `getPointsEach` method, which is defined in the `PassFailExam` class. Because the `exam` variable knows only about methods in the `GradedActivity` class, it cannot execute this method.

## Polymorphism and Dynamic Binding

When a superclass variable references a subclass object, a potential problem

exists. What if the subclass has overridden a method in the superclass, and the variable makes a call to that method? Does the variable call the superclass's version of the method, or the subclass's version? For example, look at the following code:

```
GradedActivity exam = new PassFailActivity(60);
exam.setScore(70);
System.out.println(exam.getGrade());
```

Recall that the `PassFailActivity` class inherits from the `GradedActivity` class, and it overrides the `getGrade` method. When the last statement calls the `getGrade` method, does it call the `GradedActivity` class's version (which returns "A", "B", "C", "D", or "F") or does it call the `PassFailActivity` class's version (which returns "P" or "F")?

Recall from [Chapter 6](#) the process of matching a method call with the correct method definition is known as binding. Java performs *dynamic binding* or *late binding* when a variable contains a polymorphic reference. This means the Java Virtual Machine determines at runtime which method to call, depending on the type of object that the variable references. So, it is the object's type that determines which method is called, not the variable's type. In this case, the `exam` variable references a `PassFailActivity` object, so the `PassFailActivity` class's version of the `getGrade` method is called. The last statement in this code will display a grade of P.

The program in [Code Listing 9-25](#) demonstrates polymorphic behavior. It declares an array of `GradedActivity` variables, then assigns the addresses of objects of various types to the elements of the array.

## Code Listing 9-25 (`Polymorphic.java`)

```
1 /**
2 * This program demonstrates polymorphic behavior.
3 */
4
5 public class Polymorphic
```

```

6 {
7 public static void main(String[] args)
8 {
9 // Create an array of GradedActivity references.
10 GradedActivity[] tests = new GradedActivity[3];
11
12 // The first test is a regular exam with a
13 // numeric score of 95.
14 tests[0] = new GradedActivity();
15 tests[0].setScore(95);
16
17 // The second test is a pass/fail test. The
18 // student missed 5 out of 20 questions, and the
19 // minimum passing grade is 60.
20 tests[1] = new PassFailExam(20, 5, 60);
21
22 // The third test is the final exam. There were
23 // 50 questions and the student missed 7.
24 tests[2] = new FinalExam(50, 7);
25
26 // Display the grades.
27 for (int index = 0; index < tests.length; index++)
28 {
29 System.out.println("Test " + (index+ 1) + ": " +
30 "score " + tests[index].getScore() +
31 ", grade " + tests[index].getGrade());
32 }
33 }
34 }
```

## Program Output

```

Test 1: score 95.0, grade A
Test 2: score 75.0, grade P
Test 3: score 86.0, grade B
```

You can also use parameters to polymorphically accept arguments to methods. For example, look at the following method:

```

public static void displayGrades(GradedActivity g)
{
 System.out.println("Score " + g.getScore() +
 ", grade " + g.getGrade());
}
```

This method's parameter, g, is a `GradedActivity` variable. But, it can be used to accept arguments of any type that inherits from `GradedActivity`. For example, the following code passes objects of the `FinalExam`, `PassFailActivity`, and `PassFailExam` classes to the method:

```
GradedActivity exam1 = new FinalExam(50, 7);
GradedActivity exam2 = new PassFailActivity(70);
GradedActivity exam3 = new PassFailExam(100, 10, 70);
displayGrades(exam1); // Pass a FinalExam object.
displayGrades(exam2); // Pass a PassFailActivity object.
displayGrades(exam3); // Pass a PassFailExam object.
```

## The “Is-a” Relationship Does Not Work in Reverse

It is important to note that the “is-a” relationship does not work in reverse. Although the statement “a final exam is a graded activity” is true, the statement “a graded activity is a final exam” is not true. This is because not all graded activities are final exams. Likewise, not all `GradedActivity` objects are `FinalExam` objects. So, the following code will not work:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = activity; // ERROR!
```

You cannot assign the address of a `GradedActivity` object to a `FinalExam` variable. This makes sense because `FinalExam` objects have capabilities that go beyond those of a `GradedActivity` object. Interestingly, the Java compiler will let you make such an assignment if you use a type cast, as shown here:

```
GradedActivity activity = new GradedActivity();
FinalExam exam = (FinalExam) activity; // Will compile but not ru
```

But, the program will crash when the assignment statement executes.

## The instanceof Operator

There is an operator in Java named `instanceof` that you can use to determine whether an object is an instance of a particular class. Here is the general form of an expression that uses the `instanceof` operator:

```
refVar instanceof ClassName
```

In the general form, `refVar` is a reference variable, and `ClassName` is the name of a class. This is the form of a boolean expression that will return true if the object referenced by `refVar` is an instance of `ClassName`.

Otherwise, the expression returns false. For example, the `if` statement in the following code determines whether the reference variable `activity` references a `GradedActivity` object:

```
GradedActivity activity = new GradedActivity();
if (activity instanceof GradedActivity)
 System.out.println("Yes, activity is a GradedActivity.");
else
 System.out.println("No, activity is not a GradedActivity.");
```

This code will display "Yes, activity is a GradedActivity."

The `instanceof` operator understands the “is-a” relationship that exists when a class inherits from another class. For example, look at the following code:

```
FinalExam exam = new FinalExam(20, 2);
if (exam instanceof GradedActivity)
 System.out.println("Yes, exam is a GradedActivity.");
else
 System.out.println("No, exam is not a GradedActivity.");
```

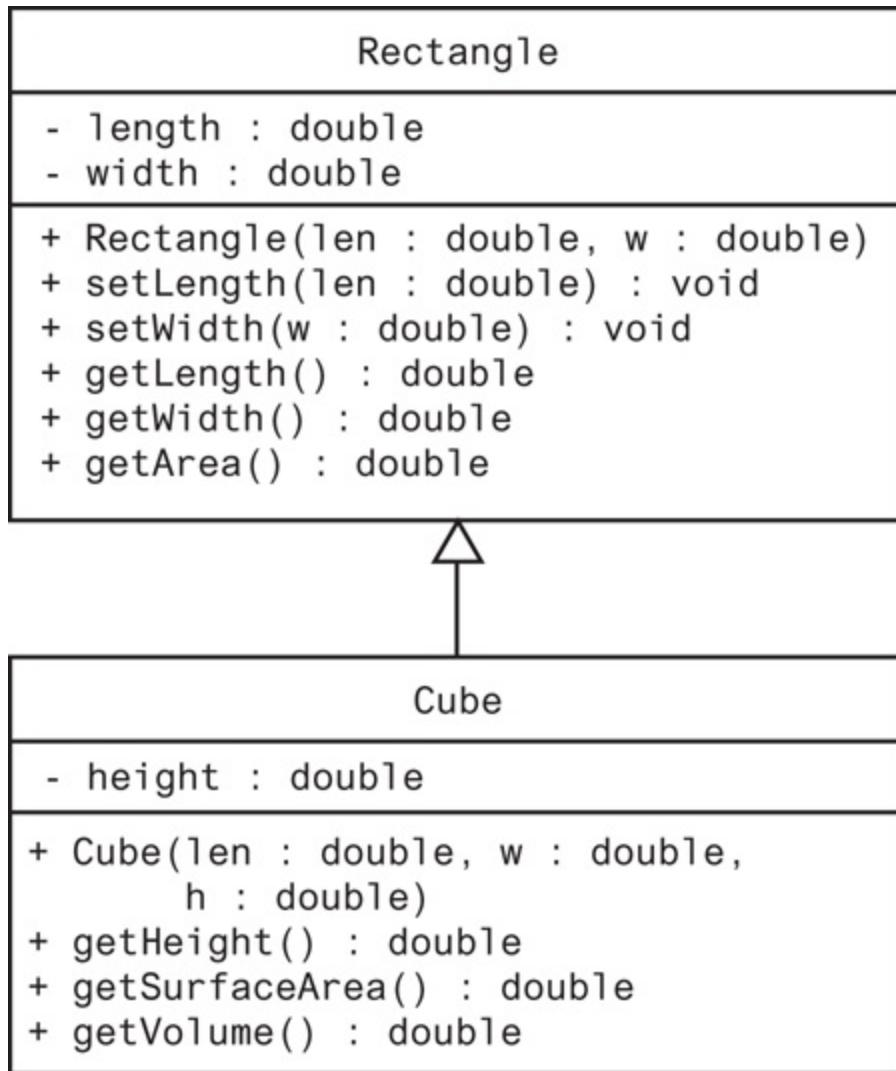
Even though the object referenced by `exam` is a `FinalExam` object, this code will display "Yes, exam is a GradedActivity." The `instanceof` operator returns true because `FinalExam` is a subclass of `GradedActivity`.



## Checkpoint

1. 9.18 Recall the `Rectangle` and `Cube` classes discussed earlier, as shown in [Figure 9-14](#).

# Figure 9-14 Rectangle and Cube classes



[Figure 9-14 Full Alternative Text](#)

1. Is the following statement legal or illegal? If it is illegal, why?

```
Rectangle r = new Cube(10, 12, 5);
```

2. If you determined that the statement in Part a is legal, are the following statements legal or illegal? (Indicate legal or illegal for

each statement.)

```
System.out.println(r.getLength());
System.out.println(r.getWidth());
System.out.println(r.getHeight());
System.out.println(r.getSurfaceArea());
```

3. Is the following statement legal or illegal? If it is illegal, why?

```
Cube c = new Rectangle(10, 12);
```

# 9.8 Abstract Classes and Abstract Methods

## Concept:

An abstract class is not instantiated, but other classes inherit from it. An abstract method has no body and must be overridden in a subclass.

An *abstract method* is a method that appears in a superclass, but expects to be overridden in a subclass. An abstract method has only a header and no body. Here is the general format of an abstract method header:

```
AccessSpecifier abstract ReturnType MethodName(ParameterList);
```

Notice the key word `abstract` appears in the header, and the header ends with a semicolon. There is no body for the method. Here is an example of an abstract method header:

```
public abstract void setValue(int value);
```

When an abstract method appears in a class, the method must be overridden in a subclass. If a subclass fails to override the method, an error will result. Abstract methods are used to ensure that a subclass implements the method.

When a class contains an abstract method, you cannot create an instance of the class. Abstract methods are commonly used in abstract classes. An *abstract class* is not instantiated itself, but serves as a superclass for other classes. The abstract class represents the generic or abstract form of all the classes that inherit from it.

For example, consider a factory that manufactures airplanes. The factory does not make a generic airplane, but makes three specific types of airplanes: two

different models of prop-driven planes, and one commuter jet model. The computer software that catalogs the planes might use an abstract class named Airplane. That class has members representing the common characteristics of all airplanes. In addition, the software has classes for each of the three specific airplane models the factory manufactures. These classes all inherit from the Airplane class, and they have members representing the unique characteristics of each type of plane. The Airplane class is never instantiated, but is used as a superclass for the other classes.

A class becomes abstract when you place the `abstract` key word in the class definition. Here is the general format:

```
public abstract class ClassName
```

For example, look at the following abstract class `Student` shown in [Code Listing 9-26](#). It holds data common to all students, but does not hold all the data needed for students of specific majors.

## Code Listing 9-26 (Student.java)

```
1 /**
2 * The Student class is an abstract class that holds general
3 * data about a student. Classes representing specific types
4 * of students should inherit from this class.
5 */
6
7 public abstract class Student
8 {
9 private String name; // Student name
10 private String idNumber; // Student ID
11 private int yearAdmitted; // Year student was admitted
12
13 /**
14 * The Constructor accepts as arguments the student's
15 * name, ID number, and the year admitted.
16 */
17
18 public Student(String n, String id, int year)
19 {
20 name = n;
21 idNumber = id;
```

```
22 yearAdmitted = year;
23 }
24
25 /**
26 * toString method
27 */
28
29 public String toString()
30 {
31 String str;
32
33 str = "Name: " + name +
34 "\nID Number: " + idNumber +
35 "\nYear Admitted: " + yearAdmitted;
36
37 return str;
38 }
39
40 /**
41 * The getRemainingHours method is abstract.
42 * It must be overridden in a subclass.
43 */
44
45 public abstract int getRemainingHours();
46 }
```

The Student class contains fields for storing a student's name, ID number, and year admitted. It also has a constructor, a `toString` method, and an abstract method named `getRemainingHours`.

This abstract method must be overridden in classes that inherit from the Student class. The idea behind this method is for it to return the number of hours remaining for a student to take in his or her major. It was made abstract because this class is intended to be the base for other classes that represent students of specific majors. For example, a `CompSciStudent` class might hold the data for a computer science student, and a `BiologyStudent` class might hold the data for a biology student. Computer science students must take courses in different disciplines than those taken by biology students. It stands to reason that the `CompSciStudent` class will calculate the number of hours remaining to be taken in a different manner than the `BiologyStudent` class. Let's look at an example of the `CompSciStudent` class, which is shown in [Code Listing 9-27](#).

# Code Listing 9-27

## (CompSciStudent.java)

```
1 /**
2 * This class holds data for a computer science student.
3 */
4
5 public class CompSciStudent extends Student
6 {
7 // Constants for the math, computer science and
8 // general education hours required for graduation.
9 private final int MATH_HOURS = 20,
10 CS_HOURS = 40,
11 GEN_ED_HOURS = 60;
12
13 private int mathHours, // Math hours taken
14 csHours, // Comp. sci. hours taken
15 genEdHours; // General ed hours taken
16
17 /**
18 * The Constructor accepts as arguments the student's
19 * name, ID number, and the year admitted.
20 */
21
22 public CompSciStudent(String n, String id, int year)
23 {
24 super(n, id, year);
25 }
26
27 /**
28 * The setMathHours method accepts a value for
29 * the number of math hours taken.
30 */
31
32 public void setMathHours(int math)
33 {
34 mathHours = math;
35 }
36
37 /**
38 * The setCsHours method accepts a value for
39 * the number of computer science hours taken.
40 */
41
```

```
42 public void setCsHours(int cs)
43 {
44 csHours = cs;
45 }
46
47 /**
48 * The setGenEdHours method accepts a value for
49 * the number of general education hours taken.
50 */
51
52 public void setGenEdHours(int genEd)
53 {
54 genEdHours = genEd;
55 }
56
57 /**
58 * toString method
59 */
60
61 @Override
62 public String toString()
63 {
64 String str; // To hold a string
65
66 // Create a string representing this computer
67 // science student's hours taken.
68 str = super.toString() +
69 "\nMajor: Computer Science" +
70 "\nMath Hours Taken: " + mathHours +
71 "\nComputer Science Hours Taken: " + csHours +
72 "\nGeneral Ed Hours Taken: " + genEdHours;
73
74 // Return the string.
75 return str;
76 }
77
78 /**
79 * The getRemainingHours method returns the
80 * the number of hours remaining to be taken.
81 */
82
83 @Override
84 public int getRemainingHours()
85 {
86 int reqHours, // Total required hours
87 remainingHours; // Remaining hours
88
89 // Calculate the total required hours.
```

```

90 reqHours = MATH_HOURS + CS_HOURS + GEN_ED_HOURS;
91
92 // Calculate the remaining hours.
93 remainingHours = reqHours - (mathHours + csHours +
94 genEdHours);
95
96 // Return the remaining hours.
97 return remainingHours;
98 }
99 }
```

The `CompSciStudent` class, which inherits from the `Student` class, declares the following `final` integer fields in lines 9 through 11: `MATH_HOURS`, `CS_HOURS`, and `GEN_ED_HOURS`. These fields hold the required number of math, computer science, and general education hours for a computer science student. It also declares the following fields in lines 13 through 15: `mathHours`, `csHours`, and `genEdHours`. These fields hold the number of math, computer science, and general education hours taken by the student. Mutator methods are provided to store values in these fields. In addition, the class overrides the `toString` method and the abstract `getRemainingHours` method. The program in [Code Listing 9-28](#) demonstrates the class.

## Code Listing 9-28 (`CompSciStudentDemo.java`)

```

1 /**
2 * This program demonstrates the CompSciStudent class.
3 */
4
5 public class CompSciStudentDemo
6 {
7 public static void main(String[] args)
8 {
9 // Create a CompSciStudent object.
10 CompSciStudent csStudent =
11 new CompSciStudent("Jennifer Haynes",
12 "167W98337", 2018);
13
14 // Store values for Math, CS, and General Ed hours.
15 csStudent.setMathHours(12);
```

```
16 csStudent.setCsHours(20);
17 csStudent.setGenEdHours(40);
18
19 // Display the student's data.
20 System.out.println(csStudent);
21
22 // Display the number of remaining hours.
23 System.out.println("Hours remaining: " +
24 csStudent.getRemainingHours());
25 }
26 }
```

## Program Output

```
Name: Jennifer Haynes
ID Number: 167W98337
Year Admitted: 2018
Major: Computer Science
Math Hours Taken: 12
Computer Science Hours Taken: 20
General Ed Hours Taken: 40
Hours remaining: 48
```

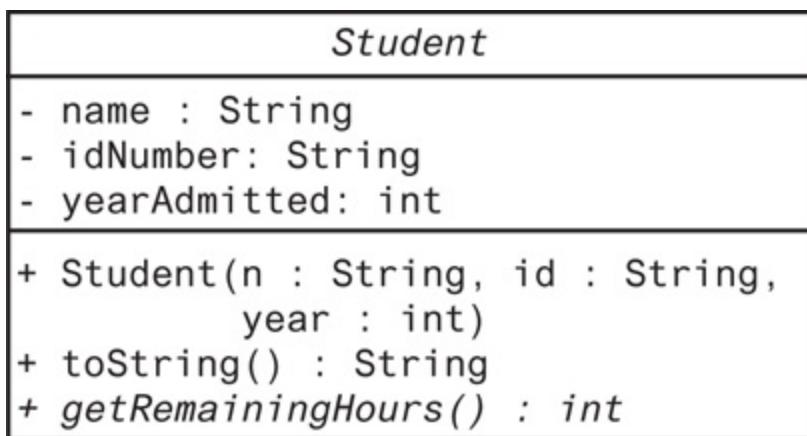
Remember the following points about abstract methods and classes:

- Abstract methods and abstract classes are defined with the `abstract` key word.
- Abstract methods have no body, and their header must end with a semicolon.
- An abstract method must be overridden in a subclass.
- When a class contains an abstract method, it cannot be instantiated. It must serve as a superclass.
- An abstract class cannot be instantiated. It must serve as a superclass.

# Abstract Classes in UML

Abstract classes are drawn like regular classes in UML, except the name of the class and the names of abstract methods are shown in italics. For example, [Figure 9-15](#) shows a UML diagram for the `Student` class.

## Figure 9-15 UML diagram for the `Student` class



[Figure 9-15 Full Alternative Text](#)



## Checkpoint

1. 9.19 What is the purpose of an abstract method?
2. 9.20 If a subclass extends a superclass with an abstract method, what must you do in the subclass?
3. 9.21 What is the purpose of an abstract class?
4. 9.22 If a class is defined as abstract, what can you not do with the class?

# 9.9 Interfaces

## Concept:

An interface specifies behavior for a class.

In its simplest form, an interface is like a class that contains only abstract methods. An interface cannot be instantiated. Instead, it is *implemented* by other classes. When a class implements an interface, the class must override the methods that are specified by the interface.

An interface looks similar to a class, except the key word `interface` is used instead of the key word `class`, and the methods that are specified in an interface do not usually bodies, only headers that are terminated by semicolons. Here is the general format of an interface definition:

```
public interface InterfaceName
{
 (Method headers . . .)
}
```

[Code Listing 9-29](#) shows an example of an interface named `Displayable`. In line 3, the interface specifies a `void` method named `display()`.

## Code Listing 9-29 (`Displayable.java`)

```
1 public interface Displayable
2 {
3 void display();
4 }
```

Notice the `display` method header in line 3 does not have an access specifier. This is because all methods in an interface are implicitly `public`. You can optionally write `public` in the method header, but most programmers leave it out because all interface methods must be `public`.

Any class that implements the `Displayable` interface shown in [Code Listing 9-29](#) *must* provide an implementation of the `display` method (with the exact signatures specified by the interface, and with the same return type). The `Person` class shown in [Code Listing 9-30](#) is an example.

## Code Listing 9-30 (Person.java)

```
1 public class Person implements Displayable
2 {
3 private String name;
4
5 // Constructor
6 public Person(String n)
7 {
8 name = n;
9 }
10
11 // display method
12 public void display()
13 {
14 System.out.println("My name is " + name);
15 }
16 }
```

When you want a class to implement an interface, you use the `implements` key word in the class header. Notice in line 1 of [Code Listing 9-30](#), the `Person` class header ends with the clause `implements Displayable`. Because the `Person` class implements the `Displayable` interface, it must provide an implementation of the interface's `display` method. This is done in lines 12 through 15 of the `Person` class. The program in [Code Listing 9-31](#) demonstrates the `Person` class.

# Code Listing 9-31

## (InterfaceDemo.java)

```
1 /**
2 * This program demonstrates a class that implements
3 * the Displayable interface.
4 */
5
6 public class InterfaceDemo
7 {
8 public static void main(String[] args)
9 {
10 // Create an instance of the Person class.
11 Person p = new Person("Antonio");
12
13 // Call the object's display method.
14 p.display();
15 }
16 }
```

### Program Output

My name is Antonio

## An Interface Is a Contract

When a class implements an interface, it is agreeing to provide all of the methods that are specified by the interface. It is often said that an interface is like a “contract,” and when a class implements an interface, it must adhere to the contract.

For example, [Code Listing 9-32](#) shows an interface named `Relatable`, which is intended to be used with the `GradedActivity` class presented earlier. This interface has three method headers: `equals`, `isGreater`, and `isLess`. Notice that each method accepts a `GradedActivity` object as its argument.

## Code Listing 9-32 (Relatable.java)

```
1 /**
2 * Relatable interface
3 */
4
5 public interface Relatable
6 {
7 boolean equals(GradedActivity g);
8 boolean isGreater(GradedActivity g);
9 boolean isLess(GradedActivity g);
10 }
```

You might have guessed that the `Relatable` interface is named “`Relatable`” because it specifies methods that, presumably, make relational comparisons with `GradedActivity` objects. The intent is to make any class that implements this interface “relatable” with `GradedActivity` objects by ensuring that it has an `equals`, an `isGreater`, and an `isLess` method that perform relational comparisons. But, the interface specifies only the signatures for these methods, not what the methods should do. Although the programmer of a class that implements the `Relatable` interface can choose what those methods do, he or she should provide methods that comply with this intent.

[Code Listing 9-33](#) shows the code for the `FinalExam3` class, which implements the `Relatable` interface. The `equals`, `isGreater`, and `isLess` methods compare the calling object with the object passed as an argument. The program in [Code Listing 9-34](#) demonstrates the class.

## Code Listing 9-33 (FinalExam3.java)

```
1 /**
2 * This class determines the grade for a final exam.
3 */
4
5 public class FinalExam3 extends GradedActivity implements Rela
```

```
6 {
7 private int numQuestions; // Number of questions
8 private double pointsEach; // Points for each question
9 private int numMissed; // Questions missed
10
11 /**
12 * The constructor sets the number of questions on the
13 * exam and the number of questions missed.
14 */
15
16 public FinalExam3(int questions, int missed)
17 {
18 double numericScore; // To hold a numeric score
19
20 // Set the numQuestions and numMissed fields.
21 numQuestions = questions;
22 numMissed = missed;
23
24 // Calculate the points for each question and
25 // the numeric score for this exam.
26 pointsEach = 100.0 / questions;
27 numericScore = 100.0 - (missed * pointsEach);
28
29 // Call the inherited setScore method to
30 // set the numeric score.
31 setScore(numericScore);
32 }
33
34 /**
35 * The getPointsEach method returns the number of
36 * points each question is worth.
37 */
38
39 public double getPointsEach()
40 {
41 return pointsEach;
42 }
43
44 /**
45 * The getNumMissed method returns the number of
46 * questions missed.
47 */
48
49 public int getNumMissed()
50 {
51 return numMissed;
52 }
```

```
53
54 /**
55 * The equals method compares the calling object
56 * to the argument object for equality.
57 */
58
59 public boolean equals(GradedActivity g)
60 {
61 boolean status;
62
63 if (this.getScore() == g.getScore())
64 status = true;
65 else
66 status = false;
67
68 return status;
69 }
70
71 /**
72 * The isGreater method determines whether the calling
73 * object is greater than the argument object.
74 */
75
76 public boolean isGreater(GradedActivity g)
77 {
78 boolean status;
79
80 if (this.getScore() > g.getScore())
81 status = true;
82 else
83 status = false;
84
85 return status;
86 }
87
88 /**
89 * The isLess method determines whether the calling
90 * object is less than the argument object.
91 */
92
93 public boolean isLess(GradedActivity g)
94 {
95 boolean status;
96
97 if (this.getScore() < g.getScore())
98 status = true;
99 else
```

```
100 status = false;
101
102 return status;
103 }
104 }
```

## Code Listing 9-34

### (RelatableExams.java)

```
1 /**
2 * This program demonstrates the FinalExam3 class which
3 * implements the Relatable interface.
4 */
5
6 public class RelatableExams
7 {
8 public static void main(String[] args)
9 {
10 // Exam #1 had 100 questions and the student
11 // missed 20 questions.
12 FinalExam3 exam1 = new FinalExam3(100, 20);
13
14 // Exam #2 had 100 questions and the student
15 // missed 30 questions.
16 FinalExam3 exam2 = new FinalExam3(100, 30);
17
18 // Display the exam scores.
19 System.out.println("Exam 1: " + exam1.getScore());
20 System.out.println("Exam 2: " + exam2.getScore());
21
22 // Compare the exam scores.
23 if (exam1.equals(exam2))
24 System.out.println("The exam scores are equal.");
25
26 if (exam1.isGreater(exam2))
27 System.out.println("The Exam 1 score is the highest.");
28
29 if (exam1.isLess(exam2))
30 System.out.println("The Exam 1 score is the lowest.");
31 }
32 }
```

## Program Output

```
Exam 1: 80.0
Exam 2: 70.0
The Exam 1 score is the highest.
```

# Fields in Interfaces

An interface can contain field declarations, but all fields in an interface are treated as `final` and `static`. Because they automatically become `final`, you must provide an initialization value. For example, look at the following interface definition:

```
public interface Doable
{
 int FIELD1 = 1,
 FIELD2 = 2;
 (Method headers . . .)
}
```

In this interface, `FIELD1` and `FIELD2` are `final static int` variables. Any class that implements this interface has access to these variables.

# Implementing Multiple Interfaces

You might be wondering why we need both abstract classes and interfaces because they are so similar to each other. The reason is that a class can directly inherit from only one superclass, but Java allows a class to implement multiple interfaces. When a class implements multiple interfaces, it must provide the methods specified by all of them.

To specify multiple interfaces in a class definition, simply list the names of the interfaces, separated by commas, after the `implements` key word. Here is the first line of an example of a class that implements multiple interfaces:

```
public class MyClass implements Interface1,
 Interface2,
```

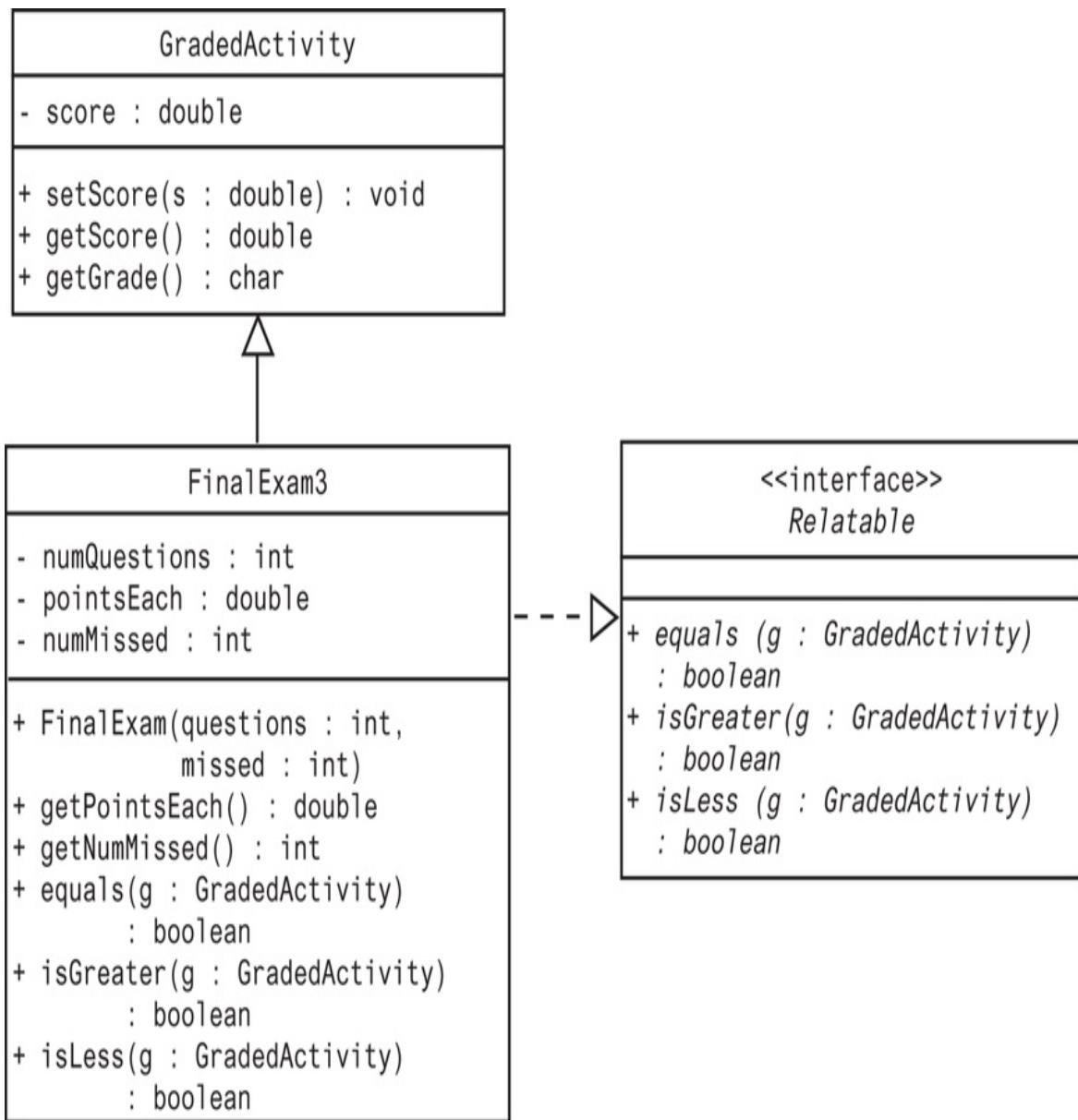
Interface3

This class implements three interfaces: Interface1, Interface2, and Interface3.

## Interfaces in UML

In a UML diagram, an interface is drawn like a class, except the interface name and the method names are italicized, and the <<interface>> tag is shown above the interface name. The relationship between a class and an interface is known as a *realization relationship* (the class realizes the interfaces). You show a realization relationship in a UML diagram by connecting a class and an interface with a dashed line that has an open arrowhead at one end. The arrowhead points to the interface. This depicts the realization relationship. [Figure 9-16](#) shows a UML diagram depicting the relationships among the GradedActivity class, the FinalExam3 class, and the Relatable interface.

### Figure 9-16 Realization relationship in a UML diagram



[Figure 9-16 Full Alternative Text](#)

## Default Methods

Beginning in Java 8, interfaces may have *default methods*. A default method is an interface method that has a body. [Code Listing 9-35](#) shows another version of the `Displayable` interface, in which the `display` method is a default method.

# Code Listing 9-35 (Displayable.java)

```
1 public interface Displayable
2 {
3 default void display()
4 {
5 System.out.println("This is the default display method.");
6 }
7 }
```

Notice in line 3 the method header begins with the key word `default`. This is required for an interface method that has a body. When a class implements an interface with a default method, the class can override the default method, but it is not required to. For example, the `Person` class shown in [Code Listing 9-36](#) implements the `Displayable` interface, but does not override the `display` method. The program in [Code Listing 9-37](#) instantiates the `Person` class, and calls the `display` method. As you can see from the program output, the code in the interface's default method is executed.

# Code Listing 9-36 (Person.java)

```
1 /**
2 * This class implements the Displayable
3 * interface, but does not override the
4 * default display method.
5 */
6
7 public class Person implements Displayable
8 {
9 private String name;
10
11 // Constructor
12 public Person(String n)
13 {
14 name = n;
15 }
16 }
```

# Code Listing 9-37 (InterfaceDemoDefaultMethod.java)

```
1 /**
2 * This program demonstrates a class that implements
3 * the Displayable interface (with a default method).
4 */
5
6 public class InterfaceDemoDefaultMethod
7 {
8 public static void main(String[] args)
9 {
10 // Create an instance of the Person class.
11 Person p = new Person("Antonio");
12
13 // Call the object's display method.
14 p.display();
15 }
16 }
```

## Program Output

This is the default display method.



## Note:

One of the benefits of having default methods is that they allow you to add new methods to an existing interface without causing errors in the classes that already implement the interface. Prior to Java 8, when you added a new method header to an existing interface, all of the classes that already implement that interface had to be rewritten to override the new method. Now you can add default methods to an interface, and if an existing class (that implements the interface) does not need the new method, you do not have to rewrite the class.

# Polymorphism and Interfaces

Just as you can create reference variables of a class type, Java allows you to create reference variables of an interface type. An interface reference variable can reference any object that implements that interface, regardless of its class type. This is another example of polymorphism. For example, look at the `RetailItem` interface in [Code Listing 9-38](#).

## Code Listing 9-38 (`RetailItem.java`)

```
1 /**
2 * RetailItem interface
3 */
4
5 public interface RetailItem
6 {
7 public double getRetailPrice();
8 }
```

This interface specifies only one method: `getRetailPrice`. Both the `CompactDisc` and `DvdMovie` classes, shown in [Code Listings 9-39](#) and [9-40](#), implement this interface.

## Code Listing 9-39 (`CompactDisc.java`)

```
1 /**
2 * Compact Disc class
3 */
4
5 public class CompactDisc implements RetailItem
6 {
7 private String title; // The CD's title
8 private String artist; // The CD's artist
```

```
9 private double retailPrice; // The CD's retail price
10
11 /**
12 * Constructor
13 */
14
15 public CompactDisc(String cdTitle, String cdArtist, double c
16 {
17 title = cdTitle;
18 artist = cdArtist;
19 retailPrice = cdPrice;
20 }
21
22 /**
23 * getTitle method
24 */
25
26 public String getTitle()
27 {
28 return title;
29 }
30
31 /**
32 * getArtist method
33 */
34
35 public String getArtist()
36 {
37 return artist;
38 }
39
40 /**
41 * getRetailPrice method (Required by the RetailItem
42 * interface)
43 */
44
45 public double getRetailPrice()
46 {
47 return retailPrice;
48 }
49 }
```

## Code Listing 9-40 (DvdMovie.java)

```
1 /**
2 * DvdMovie class
3 */
4
5 public class DvdMovie implements RetailItem
6 {
7 private String title; // The DVD's title
8 private int runningTime; // Running time in minutes
9 private double retailPrice; // The DVD's retail price
10
11 /**
12 * Constructor
13 */
14
15 public DvdMovie(String dvdTitle, int runTime, double dvdPric
16 {
17 title = dvdTitle;
18 runningTime = runTime;
19 retailPrice = dvdPrice;
20 }
21
22 /**
23 * getTitle method
24 */
25
26 public String getTitle()
27 {
28 return title;
29 }
30
31 /**
32 * getRunningTime method
33 */
34
35 public int getRunningTime()
36 {
37 return runningTime;
38 }
39
40 /**
41 * getRetailPrice method (Required by the RetailItem
42 * interface)
43 */
44
45 public double getRetailPrice()
46 {
47 return retailPrice;
48 }
```

```
49 }
```

Because they implement the `RetailItem` interface, objects of these classes can be referenced by a `RetailItem` reference variable. The following code demonstrates this:

```
RetailItem item1 = new CompactDisc("Songs From the Heart",
 "Billy Nelson",
 18.95);
RetailItem item2 = new DvdMovie("Planet X",
 102,
 22.95);
```

In this code, two `RetailItem` reference variables, `item1` and `item2`, are declared. The `item1` variable references a `CompactDisc` object, and the `item2` variable references a `DvdMovie` object. This is possible because both the `CompactDisc` and `DvdMovie` classes implement the `RetailItem` interface. When a class implements an interface, an inheritance relationship known as *interface inheritance* is established. Because of this inheritance relationship, a `CompactDisc` object *is a* `RetailItem`, and likewise, a `DvdMovie` object *is a* `RetailItem`. Therefore, we can create `RetailItem` reference variables and have them reference `CompactDisc` and `DvdMovie` objects.

The program in [Code Listing 9-41](#) demonstrates how an interface reference variable can be used as a method parameter.

## Code Listing 9-41 (`PolymorphicInterfaceDemo.java`)

```
1 /**
2 * This program demonstrates that an interface type may
3 * be used to create a polymorphic reference.
4 */
5
6 public class PolymorphicInterfaceDemo
7 {
8 public static void main(String[] args)
9 {
```

```

10 // Create a CompactDisc object.
11 CompactDisc cd =
12 new CompactDisc("Greatest Hits",
13 "Joe Looney Band",
14 18.95);
15 // Create a DvdMovie object.
16 DvdMovie movie =
17 new DvdMovie("Wheels of Fury",
18 137, 12.95);
19
20 // Display the CD's title.
21 System.out.println("Item #1: " +
22 cd.getTitle());
23
24 // Display the CD's price.
25 showPrice(cd);
26
27 // Display the DVD's title.
28 System.out.println("Item #2: " +
29 movie.getTitle());
30
31 // Display the DVD's price.
32 showPrice(movie);
33 }
34
35 /**
36 * The showPrice method displays the price
37 * of the RetailItem object that is passed
38 * as an argument.
39 */
40
41 private static void showPrice(RetailItem item)
42 {
43 System.out.printf("Price: $%,.2f\n", item.getRetailPrice())
44 }
45 }
```

## Program Output

```

Item #1: Greatest Hits
Price: $18.95
Item #2: Wheels of Fury
Price: $12.95
```

There are some limitations to using interface reference variables. As

previously mentioned, you cannot create an instance of an interface. The following code will cause a compiler error:

```
RetailItem item = new RetailItem(); // ERROR! Will not compile!
```

In addition, when an interface variable references an object, you can use the interface variable to call only the methods that are specified in the interface. For example, look at the following code:

```
// Reference a CompactDisc object with a RetailItem variable.
RetailItem item = new CompactDisc("Greatest Hits",
 "Joe Looney Band",
 18.95);
// Call the getRetailPrice method...
System.out.println(item.getRetailPrice()); // OK, this works.
// Attempt to call the getTitle method...
System.out.println(item.getTitle()); // ERROR! Will not compile!
```

The last line of code will not compile because the `RetailItem` interface specifies only one method: `getRetailPrice`. So, we cannot use a `RetailItem` reference variable to call any other method.<sup>1</sup>

<sup>1</sup> Actually, it is possible to cast an interface reference variable to the type of the object it references, then call methods that are members of that type. The syntax is somewhat awkward, however. The statement that causes the compiler error in the example code could be rewritten as:

```
System.out.println(((CompactDisc)item).getTitle());
```



## Checkpoint

1. 9.23 What is the purpose of an interface?
2. 9.24 How is an interface similar to an abstract class?
3. 9.25 How is an interface different from an abstract class, or any class?
4. 9.26 If an interface has fields, how are they treated?

5. 9.27 Write the first line of a class named `Customer`, which implements an interface named `Relatable`.
6. 9.28 Write the first line of a class named `Employee`, which implements interfaces named `Payable` and `Listable`.

# 9.10 Anonymous Inner Classes

## Concept:

An inner class is a class that is defined inside another class. An anonymous inner class is an inner class that has no name. An anonymous inner class must implement an interface, or extend another class.

Sometimes you need a class that is simple, and to be instantiated only once in your code. When this is the case, you can use an anonymous inner class. An *anonymous inner class* is a class that has no name. It is called an inner class because it is defined inside another class. You use the new operator to simultaneously define an anonymous inner class and create an instance of it. Here is the general syntax for instantiating and defining an anonymous inner class:

```
new ClassOrInterfaceName() {
 (Fields and methods of the anonymous class...)
}
```

The new operator is followed by the name of an existing class or interface, followed by a set of parentheses. Next, you write the body of the class, enclosed in curly braces. The expression creates an object that is an instance of a class that either extends the specified superclass, or implements the specified interface. A reference to the object is returned. (Notice you do *not* use the extends or implements key words in the expression.)

Before you look at an example, you must understand a few requirements and restrictions:

- An anonymous inner class must either implement an interface, or extend another class.

- If the anonymous inner class extends a superclass, the superclass's no-arg constructor is called when the object is created.
- An anonymous inner class must override all of the abstract methods specified by the interface it is implementing, or the superclass it is extending.
- Because an anonymous inner class's definition is written inside a method, it can access that method's local variables, but only if they are declared `final`, or if they are effectively `final`. (An effectively `final` variable is a variable whose value is never changed.) A compiler error will result if an anonymous inner class tries to use a variable that is not `final`, or not effectively `final`.

Let's look at an example of an anonymous inner class that implements an interface. Suppose we have the interface shown in [Code Listing 9-42](#).

## Code Listing 9-42 (`IntCalculator.java`)

```
1 interface IntCalculator
2 {
3 int calculate(int number);
4 }
```

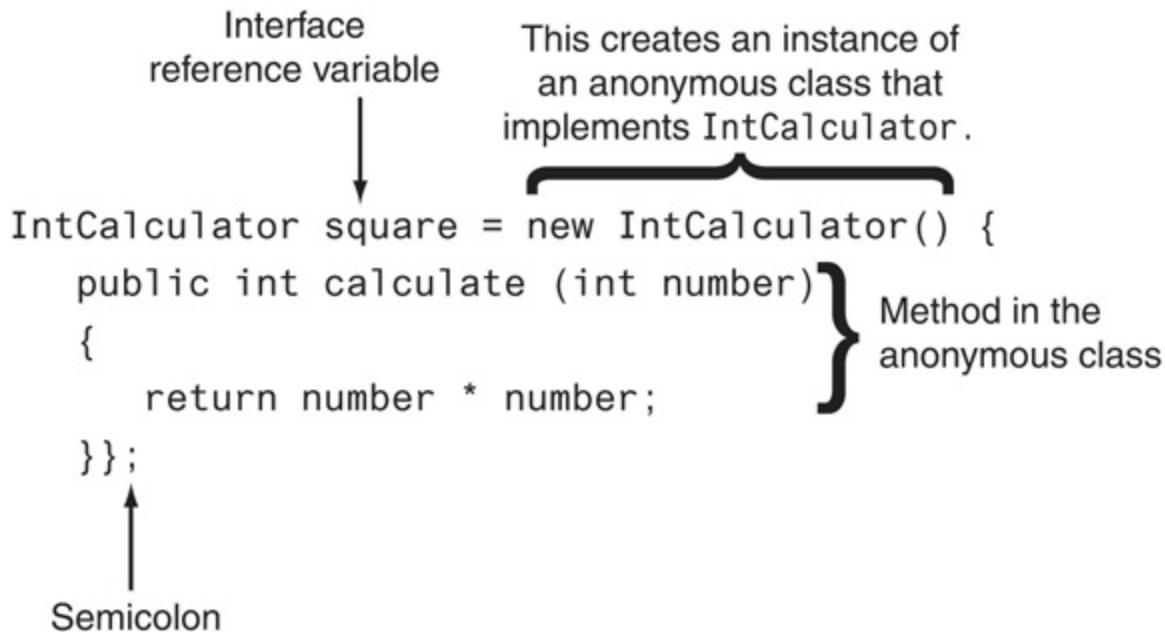
The name of the interface is `IntCalculator`, and it specifies a method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value. Suppose we want to define a class that implements the `IntCalculator` interface, and overrides the `calculate` method so that it returns the square of the argument that is passed to it. The following code snippet shows how:

```
IntCalculator square = new IntCalculator() {
 public int calculate(int number)
 {
 return number * number;
 }};

```

The first line of the code snippet declares a variable named square, that is an IntCalculator reference variable (meaning that it can refer to any object that implements IntCalculator). On the right side of the = sign, the expression new IntCalculator() creates an instance of an anonymous class that implements the IntCalculator interface. The body of the anonymous class appears next, enclosed inside curly braces. In the class body, the calculate method is overridden. Because this is a complete statement, it ends with a semicolon. [Figure 9-21](#) illustrates the different parts of the statement, and [Code Listing 9-43](#) shows a complete program that uses it.

## Figure 9-21 Creating an instance of an anonymous inner class



[Figure 9-21 Full Alternative Text](#)

## Code Listing 9-43

# (AnonymousClassDemo.java)

```
1 /**
2 * This program demonstrates an anonymous inner class
3 */
4
5 import java.util.Scanner;
6
7 public class AnonymousClassDemo
8 {
9 public static void main(String[] args)
10 {
11 int num;
12
13 // Create a Scanner object for keyboard input.
14 Scanner keyboard = new Scanner(System.in);
15
16 // Create an object that implements IntCalculator.
17 IntCalculator square = new IntCalculator() {
18 public int calculate(int number)
19 {
20 return number * number;
21 }
22
23 // Get a number from the user.
24 System.out.print("Enter an integer number: ");
25 num = keyboard.nextInt();
26
27 // Display the square of the number.
28 System.out.println("The square is " + square.calculate(num)
29 }
30 }
```

## Program Output with Example Input Shown in Bold

Enter an integer number: **5**   
The square is 25

Let's take a closer look at the program:

- Line 11 declares an `int` variable named `num`, that will be used to hold user input.

- Line 14 creates a Scanner object for keyboard input.
- Lines 17 through 21 instantiate an anonymous inner class that implements the IntCalculator interface. A variable named square is used to reference the object. In the class body, lines 18 through 21 override the calculate method to return the square of the method's argument.
- Line 24 prompts the user to enter an integer number, and line 25 reads the number from the keyboard. The number is assigned to the num variable.
- Line 28 calls the square object's calculate method, and displays the return value in a message.

# 9.11 Functional Interfaces and Lambda Expressions

## Concept:

A functional interface is an interface that has one abstract method. You can use a special type of expression, known as a lambda expression, to create an object that implements a functional interface.

Java 8 introduced two new features that work together to simplify code, particularly in situations where you might use anonymous inner classes. These new features are functional interfaces and lambda expressions. A *functional interface* is simply an interface that has one abstract method. For example, let's take another look at the `IntCalculator` interface that we previously discussed in the section on anonymous classes. Because it has only one abstract method, it is considered a functional interface. For your convenience, [Code Listing 9-44](#) shows the code for the interface.

## Code Listing 9-44 (`IntCalculator.java`)

```
1 interface IntCalculator
2 {
3 int calculate(int number);
4 }
```

The name of the interface is `IntCalculator`, and it specifies one method named `calculate`. The `calculate` method accepts an `int` argument, and returns an `int` value.

Because `IntCalculator` is a functional interface, we do not have to go to the trouble of defining a class that implements the interface. We do not even have to use an anonymous inner class. Instead, we can use a *lambda expression* to create an object that implements the interface, and overrides its abstract method.

You can think of a lambda expression as an anonymous method, or a method with no name. Like regular methods, lambda expressions can accept arguments and return values. Here is the general format of a simple lambda expression that accepts one argument, and returns a value:

*parameter -> expression*

In this general format, the lambda expression begins with a parameter variable, followed by the *lambda operator* (`->`), followed by an expression that has a value. Here is an example:

```
x -> x * x
```

The `x` that appears on the left side of the `->` operator is the name of a parameter variable, and the expression `x * x` that appears on the right side of the `->` operator is the value that is returned.

This lambda expression works like a method that has a parameter variable named `x`, and it returns the value of `x * x`.

We can use this lambda expression to create an object that implements the `IntCalculator` interface. Here is an example:

```
IntCalculator square = x -> x * x;
```

On the left side of the `=` operator, we declare an `IntCalculator` reference variable named `square`. On the right side of the `=` operator, we have a lambda expression that creates an object with the following characteristics:

- Because we are assigning the object to an `IntCalculator` reference variable, the object automatically implements the `IntCalculator` interface.

- Because the `IntCalculator` interface has only one abstract method (named `calculate`), the lambda expression will be used to implement that one method.
- The parameter `x` that is used in the lambda expression represents the argument that is passed to the `calculate` method. We do not have to specify the data type of `x` because the compiler will determine it. Because the `calculate` method (in the `IntCalculator` interface) has an `int` parameter, the `x` parameter in the lambda expression will automatically become an `int`.
- The expression `x * x` is the value that is returned from the `calculate` method.

[Code Listing 9-45](#) shows a complete program that uses the previously shown statement to create an object.

## Code Listing 9-45 (`LambdaDemo.java`)

```

1 /**
2 * This program demonstrates a simple
3 * lambda expression.
4 */
5
6 import java.util.Scanner;
7
8 public class LambdaDemo
9 {
10 public static void main(String[] args)
11 {
12 int num;
13
14 // Create a Scanner object for keyboard input.
15 Scanner keyboard = new Scanner(System.in);
16
17 // Create an object that implements IntCalculator.
18 IntCalculator square = x -> x * x;
19
20 // Get a number from the user.

```

```
21 System.out.print("Enter an integer number: ");
22 num = keyboard.nextInt();
23
24 // Display the square of the number.
25 System.out.println("The square is " + square.calculate(num))
26 }
27 }
```

## Program Output with Example Input Shown in Bold

Enter an integer number: **5**   
The square is 25

Let's take a closer look at the program:

- Line 12 declares an `int` variable named `num`, which will be used to hold user input.
- Line 15 creates a `Scanner` object for keyboard input.
- Line 18 uses a lambda expression to create an object that implements the `IntCalculator` interface. A variable named `square` is used to reference the object. The object's `calculate` method will return the square of the method's argument.
- Line 21 prompts the user to enter an integer number, and line 22 reads the number from the keyboard. The number is assigned to the `num` variable.
- Line 25 calls the `square` object's `calculate` method, and displays the return value in a message.

Lambda expressions provide a way to easily create and instantiate anonymous inner classes. If you compare [Code Listing 9-45](#) with the program shown in [Code Listing 9-43](#), you can see that the lambda expression is much more concise than the anonymous inner class declaration.

# Lambda Expressions That Do Not

# Return a Value

If a functional interface's abstract method is `void` (does not return a value), any lambda expression that you use with the interface should also be `void`. Here is an example:

```
x -> System.out.println(x);
```

This lambda expression has a parameter, `x`. When the expression is invoked, it displays the value of `x`.

# Lambda Expressions with Multiple Parameters

If a functional interface's abstract method has multiple parameters, any lambda expression that you use with the interface must also have multiple parameters. To use more than one parameter in a lambda expression, simply write a comma-separated list, then enclose the list in parentheses. Here is an example:

```
(a, b) -> a + b;
```

This lambda expression has two parameters, `a` and `b`. The expression returns the value of `a + b`.

# Lambda Expressions with No Parameters

If a functional interface's abstract method has no parameters, any lambda expression that you use with the interface must also have no parameters. Simply write a set of empty parentheses as the parameter list, as shown here:

```
() -> System.out.println();
```

When this lambda expression is invoked, it simply prints a blank line.

## Explicitly Declaring a Parameter's Data Type

You do not have to specify the data type of a lambda expression's parameter because the compiler will determine it from the interface's abstract method header. However, you can explicitly declare the data type of a parameter, if you wish. Here is an example:

```
(int x) -> x * x;
```

Note the parameter declaration (on the left side of the `->` operator) must be enclosed in parentheses. Here is another example, involving two parameters:

```
(int a, int b) -> a + b;
```

## Using Multiple Statements in the Body of a Lambda Expression

You can write multiple statements in the body of a lambda expression, but if you do, you must enclose the statements in a set of curly braces, and you must write a `return` statement if the expression returns a value. Here is an example:

```
(int x) -> {
 int a = x * 2;
 return a;
};
```

## Accessing Variables Within a

# Lambda Expression

A lambda expression can access variables that are declared in the enclosing scope, as long as those variables are `final`, or effectively `final`. An *effectively final* variable is a variable whose value is never changed, but it isn't declared with the `final` key word.

In [Code Listing 9-46](#), the `main` method uses a lambda expression that accesses a `final` variable named `factor` that is local to the `main` method.

## Code Listing 9-46 (`LambdaDemo2.java`)

```
1 /**
2 * This program demonstrates a lambda expression
3 * that uses a final local variable.
4 */
5
6 import java.util.Scanner;
7
8 public class LambdaDemo2
9 {
10 public static void main(String[] args)
11 {
12 final int factor = 10;
13 int num;
14
15 // Create a Scanner object for keyboard input.
16 Scanner keyboard = new Scanner(System.in);
17
18 // Create an object that implements IntCalculator.
19 IntCalculator multiplier = x -> x * factor;
20
21 // Get a number from the user.
22 System.out.print("Enter an integer number: ");
23 num = keyboard.nextInt();
24
25 // Display the number multiplied by 10.
26 System.out.println("Multiplied by 10, that number is " +
27 multiplier.calculate(num));
```

```
28 }
29 }
```

## Program Output with Example Input Shown in Bold

Enter an integer number: **10**   
Multiplied by 10, that number is 100

In [Code Listing 9-46](#), we could remove the `final` key word from the variable declaration in line 12, and the program would still compile and execute correctly. This is because the `factor` variable is never modified, and therefore is effectively `final`.

## 9.12 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- Attempting to directly access a private superclass member from a subclass. Private superclass members cannot be directly accessed by a method in a subclass. The subclass must call a public or protected superclass method in order to access the superclass's private members.
- Forgetting to explicitly call a superclass constructor when the superclass has no default constructor or programmer-defined no-arg constructor. When a superclass does not have a default constructor or a no-arg constructor, the subclass's constructor must explicitly call one of the constructors that the superclass does have.
- Allowing the superclass's default constructor or no-arg constructor to be implicitly called when you intend to call another superclass constructor. If a subclass's constructor does not explicitly call a superclass constructor, Java automatically calls `super()`.
- Forgetting to precede a call to an overridden superclass method with `super`. When a subclass method calls an overridden superclass method, it must precede the method call with the key word `super` and a dot (`.`). Failing to do so results in the subclass's version of the method being called.
- Forgetting a class member's access specifier. When you do not give a class member an access specifier, it is granted package access by default. This means that any method in the same package can access the member.
- Writing a body for an abstract method. An abstract method cannot have a body. It must be overridden in a subclass.
- Forgetting to terminate an abstract method's header with a semicolon.

An abstract method header does not have a body, and it must be terminated with a semicolon.

- Failing to override an abstract method. An abstract method must be overridden in a subclass.
- Overloading an abstract method instead of overriding it. Overloading is not the same as overriding. When a superclass has an abstract method, the subclass must have a method with the same signature as the abstract method.
- Trying to instantiate an abstract class. You cannot create an instance of an abstract class.
- Implementing an interface but forgetting to override all of its methods. When a class implements an interface, all of the methods specified by the interface must be overridden in the class.
- Overloading an interface method instead of overriding it. As previously mentioned, overloading is not the same as overriding. When a class implements an interface, the class must have methods with the same signature as the methods specified in the interface.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. In an inheritance relationship, this is the general class.
  1. subclass
  2. superclass
  3. derived class
  4. child class
  
2. In an inheritance relationship, this is the specialized class.
  1. superclass
  2. base class
  3. subclass
  4. parent class
  
3. This key word indicates that a class inherits from another class.
  1. derived
  2. specialized
  3. based
  4. extends
  
4. A subclass does not have access to these superclass members.

1. public
  2. private
  3. protected
  4. all of these
5. This key word refers to an object's superclass.
1. super
  2. base
  3. this
  4. parent
6. In a subclass constructor, a call to the superclass constructor must
- .
1. appear as the very first statement
  2. appear as the very last statement
  3. appear between the constructor's header and the opening brace
  4. not appear
7. The following is an explicit call to the superclass's default constructor.
1. default();
  2. class();
  3. super();
  4. base();
8. A method in a subclass having the same signature as a method in the

superclass is an example of .

1. overloading

2. overriding

3. composition

4. an error

9. A method in a subclass having the same name as a method in the superclass but a different signature is an example of .

1. overloading

2. overriding

3. composition

4. an error

10. These superclass members are accessible to subclasses and classes in the same package.

1. private

2. public

3. protected

4. all of these

11. All classes directly or indirectly inherit from this class.

1. Object

2. Super

3. Root
  4. Java
12. With this type of binding, the Java Virtual Machine determines at runtime which method to call, depending on the type of the object that a variable references.
1. static
  2. early
  3. flexible
  4. dynamic
13. When a class implements an interface, it must do which of the following?
1. Overload all of the methods listed in the interface
  2. Provide all of the nondefault methods that are listed in the interface, with the exact signatures specified
  3. Not have a constructor
  4. Be an abstract class
14. Fields in an interface are .
1. `final`
  2. `static`
  3. both `final` and `static`
  4. not allowed
15. Abstract methods must be .

1. overridden
2. overloaded
3. deleted and replaced with real methods
4. declared as private

16. Abstract classes cannot .

1. be used as superclasses
2. have abstract methods
3. be instantiated
4. have fields

17. You use the operator to define an anonymous inner class.

1. class
2. inner
3. new
4. anonymous

18. An anonymous inner class must .

1. be a superclass
2. implement an interface
3. extend a superclass
4. either b or c.

19. A functional interface is an interface with .

1. only one abstract method
  2. no abstract methods
  3. only private methods
  4. no name
20. You can use a lambda expression to instantiate an object that
- .
  - 1. has no constructor
  - 2. extends any superclass
  - 3. implements a functional interface
  - 4. does not implement an interface
21. True or False: Constructors are not inherited.
22. True or False: In a subclass, a call to the superclass constructor can be written only in the subclass constructor.
23. True or False: If a subclass constructor does not explicitly call a superclass constructor, Java will not call any of the superclass's constructors.
24. True or False: An object of a superclass can access members declared in a subclass.
25. True or False: The superclass constructor always executes before the subclass constructor.
26. True or False: When a method is declared with the `final` modifier, it must be overridden in a subclass.
27. True or False: A superclass has a member with package access. A class that is outside the superclass's package, but inherits from the superclass,

can access this member.

28. True or False: A superclass reference variable can reference an object of a class that inherits from the superclass.
29. True or False: A subclass reference variable can reference an object of the superclass.
30. True or False: When a class contains an abstract method, the class cannot be instantiated.
31. True or False: A class can implement only one interface.
32. True or False: By default, all members of an interface are public.

## Find the Error

Find the error in each of the following code segments.

```
1. // Superclass
 public class Vehicle
 {
 (Member declarations . . .)
 }
 // Subclass
 public class Car expands Vehicle
 {
 (Member declarations . . .)
 }
```

```
2. // Superclass
 public class Vehicle
 {
 private double cost;
 (Other methods . . .)
 }
 // Subclass
 public class Car extends Vehicle
 {
 public Car(double c)
 {
```

```

 cost = c;
 }
}

3. // Superclass
public class Vehicle
{
 private double cost;
 public Vehicle(double c)
 {
 cost = c;
 }
(Other methods . . .)
}
// Subclass
public class Car extends Vehicle
{
 private int passengers;
 public Car(int p)
 {
 passengers = c;
 }
(Other methods . . .)
}

4. // Superclass
public class Vehicle
{
 public abstract double getMilesPerGallon();
(Other methods . . .)
}
// Subclass
public class Car extends Vehicle
{
 private int mpg;

 public int getMilesPerGallon()
 {
 return mpg;
 }
(Other methods . . .)
}

```

# Algorithm Workbench

1. Write the first line of the definition for a Poodle class. The class should inherit from the Dog class.
2. Look at the following code which is the first line of a class definition:

```
public class Tiger extends Felis
```

In what order will the class constructors execute?

3. Write the declaration for class B. The class's members should be:
  - m, an integer. This variable should not be accessible to code outside the class or to any class that inherits from class B.
  - n, an integer. This variable should be accessible only to classes that inherit from class B or in the same package as class B.
  - setM, getM, setN, and getN. These are the mutator and accessor methods for the member variables m and n. These methods should be accessible to code outside the class.
  - calc. This is a public abstract method.

Next write the declaration for class D, which inherits from class B. The class's members should be:

- q, a double. This variable should not be accessible to code outside the class.
- r, a double. This variable should be accessible to any class that extends class D or in the same package.
- setQ, getQ, setR, and getR. These are the mutator and accessor methods for the member variables q and r. These methods should be accessible to code outside the class.
- calc. a public method that overrides the superclass's abstract calc method. This method should return the value of q times r.

4. Write the statement that calls a superclass constructor and passes the arguments `x`, `y`, and `z`.
5. A superclass has the following method:

```
public void setValue(int v)
{
 value = v;
}
```

Write a statement that can appear in a subclass that calls this method, passing 10 as an argument.

6. A superclass has the following abstract method:

```
public abstract int getValue();
```

Write an example of a `getValue` method that can appear in a subclass.

7. Write the first line of the definition for a `Stereo` class. The class should inherit from the `SoundSystem` class, and it should implement the `CDPlayable`, `TunerPlayable`, and `MP3Playable` interfaces.
8. Write an interface named `Nameable` that specifies the following methods:

```
public void setName(String n)
public String getName()
```

9. Look at the following interface:

```
public interface Computable
{
 void compute(double x);
}
```

Write a statement that uses a lambda expression to create an object that implements the `Computable` interface. The object's name should be `half`. The `half` object's `compute` method should return the value of the `x` parameter divided by 2.

# Short Answer

1. What is an “is-a” relationship?
2. A program uses two classes: Animal and Dog. Which class is the superclass and which is the subclass?
3. What is the superclass and what is the subclass in the following line?

```
public class Pet extends Dog
```
4. What is the difference between a protected class member and a private class member?
5. Can a subclass ever directly access the private members of its superclass?
6. Which constructor is called first, that of the subclass or the superclass?
7. What is the difference between overriding a superclass method and overloading a superclass method?
8. Reference variables can be polymorphic. What does this mean?
9. When does dynamic binding take place?
10. What is an abstract method?
11. What is an abstract class?
12. What are the differences between an abstract class and an interface?
13. When you instantiate an anonymous inner class, the class must do one of two things. What are they?
14. What is a functional interface?
15. What is a lambda expression?

# Programming Challenges

## 1. Employee and ProductionWorker Classes

Design a class named `Employee`. The class should keep the following information in fields:

- Employee name
- Employee number in the format XXX–L, where each X is a digit within the range 0–9, and the L is a letter within the range A–M.
- Hire date



**VideoNote** The Employee and ProductionWorker Classes Problem

Write one or more constructors and the appropriate accessor and mutator methods for the class.

Next, write a class named `ProductionWorker` that inherits from the `Employee` class. The `ProductionWorker` class should have fields to hold the following information:

- Shift (an integer)
- Hourly pay rate (a double)

The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1, and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

## 2. ShiftSupervisor Class

In a particular factory, a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that inherits from the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a field that holds the annual salary, and a field that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

## 3. TeamLeader Class

In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that inherits from the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have fields for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

## 4. Essay Class

Design an `Essay` class that inherits from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100 and is determined in the following manner:

Grammar: 30 points

Spelling: 20 points

Correct length: 20 points

Content: 30 points

Demonstrate the class in a simple program.

## 5. Course Grades

In a course, a teacher gives the following tests and assignments:

- A **lab activity** that is observed by the teacher and assigned a numeric score.
- A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
- An **essay** that is assigned a numeric score.
- A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have an array of `GradedActivity` objects as a field. The array should be named `grades`. The `grades` array should have four elements, one for each of the assignments previously described. The class should have the following methods:

`setLab`

This method should accept a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` field should reference this object.

`setPassFailExam`

This method should accept a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` field should reference this object.

`setEssay`

This method should accept an `Essay` object as its argument. (See Programming Challenge 4 for the `Essay` class. If you have not completed Programming Challenge 4, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` field should reference this object.

`setFinalExam`

This method should accept a `FinalExam` object as its argument. This object should already hold the

student's score for the final exam. Element 3 of the grades field should reference this object.

This method should return a string that contains the numeric scores and grades for each element in the grades array.

Demonstrate the class in a program.

## 6. Analyzable Interface

Modify the CourseGrades class you created in Programming Challenge 5 so that it implements the following interface:

```
public interface Analyzable
{
 double getAverage();
 GradedActivity getHighest();
 GradedActivity getLowest();
}
```

The `getAverage` method should return the average of the numeric scores stored in the grades array. The `getHighest` method should return a reference to the element of the grades array that has the highest numeric score. The `getLowest` method should return a reference to the element of the grades array that has the lowest numeric score. Demonstrate the new methods in a complete program.

## 7. Person and Customer Classes

Design a class named `Person` with fields for holding a person's name, address, and telephone number. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields.

Next, design a class named `Customer`, which inherits from the `Person` class. The `Customer` class should have a field for a customer number and a boolean field indicating whether the customer wishes to be on a mailing list. Write one or more constructors, and the appropriate mutator and accessor methods, for the class's fields. Demonstrate an object of the `Customer` class in a simple program.

## 8. PreferredCustomer Class

A retail store has a preferred customer plan where customers can earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store, as follows:

- When a preferred customer spends \$500, he or she gets a 5 percent discount on all future purchases.
- When a preferred customer spends \$1,000, he or she gets a 6 percent discount on all future purchases.
- When a preferred customer spends \$1,500, he or she gets a 7 percent discount on all future purchases.
- When a preferred customer spends \$2,000 or more, he or she gets a 10 percent discount on all future purchases.

Design a class named `PreferredCustomer`, which inherits from the `Customer` class you created in Programming Challenge 7. The `PreferredCustomer` class should have fields for the amount of the customer's purchases and the customer's discount level. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate the class in a simple program.

## 9. BankAccount and SavingsAccount Classes

Design an abstract class named `BankAccount` to hold the following data for a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate

- Monthly service charges

The class should have the following methods:

|                |                                                                                                                                                                                                                            |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Constructor    | The constructor should accept arguments for the balance and annual interest rate.                                                                                                                                          |
| deposit        | A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits.                          |
| withdraw       | A method that accepts an argument for the amount of the withdrawal. The method should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals.                     |
|                | A method that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:                                         |
| calcInterest   | $\text{Monthly Interest Rate} = (\text{Annual Interest Rate}/12)$ $\text{Monthly Interest} = \text{Balance} * \text{Monthly Interest Rate}$ $\text{Balance} = \text{Balance} + \text{Monthly Interest}$                    |
| monthlyProcess | A method that subtracts the monthly service charges from the balance, calls the calcInterest method, then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero. |

Next, design a `SavingsAccount` class that extends the `BankAccount` class. The `SavingsAccount` class should have a status field to represent an active or inactive account. If the balance of a savings account falls below \$25, it becomes inactive. (The status field could be a boolean

variable.) No more withdrawals can be made until the balance is raised above \$25, at which time the account becomes active again. The savings account class should have the following methods:

withdraw

A method that determines whether the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the superclass version of the method.

deposit

A method that determines whether the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above \$25, the account becomes active again. The deposit is then made by calling the superclass version of the method.

monthlyProcess

Before the superclass method is called, this method checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of \$1 for each withdrawal above 4 is added to the superclass field that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below \$25, the account becomes inactive.)

## 10. Ship, CruiseShip, and CargoShip Classes

Design a `Ship` class with the following members:

- A field for the name of the ship (a string)
- A field for the year that the ship was built (a string)
- A constructor and appropriate accessors and mutators
- A `toString` method that displays the ship's name and the year it was built

Design a `CruiseShip` class that extends the `Ship` class. The `CruiseShip` class should have the following members:

- A field for the maximum number of passengers (an `int`)
- A constructor and appropriate accessors and mutators
- A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following members:

- A field for the cargo capacity in tonnage (an `int`)
- A constructor and appropriate accessors and mutators
- A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has a `Ship` array. Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements. The program should then step through the array, calling each object's `toString` method. (See [Code Listing 9-25](#) as an example.)

# **Chapter 10 Exceptions and Advanced File I/O**

## **Topics**

1. [10.1 Handling Exceptions](#)
2. [10.2 Throwing Exceptions](#)
3. [10.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization](#)
4. [10.4 Common Errors to Avoid](#)

# 10.1 Handling Exceptions

## Concept:

An exception is an object that is generated as the result of an error or an unexpected event. To prevent exceptions from crashing your program, you must write code that detects and handles them.



### VideoNote Handling Exceptions

There are many error conditions that can occur while a Java application is running that will cause it to halt execution. By now, you have probably experienced this many times. For example, look at the program in [Code Listing 10-1](#). This program attempts to read beyond the bounds of an array.

## Code Listing 10-1 (BadArray.java)

```
1 /**
2 * This program causes an error and crashes.
3 */
4
5 public class BadArray
6 {
7 public static void main(String[] args)
8 {
9 // Create an array with three elements.
10 int[] numbers = { 1, 2, 3 };
11
12 // Attempt to read beyond the bounds
13 // of the array.
14 for (int index = 0; index <= 3; index++)
15 System.out.println(numbers[index]);
```

```
16 }
17 }
```

## Program Output

```
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
at BadArray.main(BadArray.java:15)
```

The numbers array in this program has only three elements, with the subscripts 0 through 2. The program crashes when it tries to read the element at numbers[3] and displays the error message you see at the end of the program output. This message indicates that an exception occurred, and it gives some information about it. An *exception* is an object that is generated in memory as the result of an error or an unexpected event. When an exception is generated, it is said to have been “thrown.” Unless an exception is detected by the application and dealt with, it causes the application to halt.

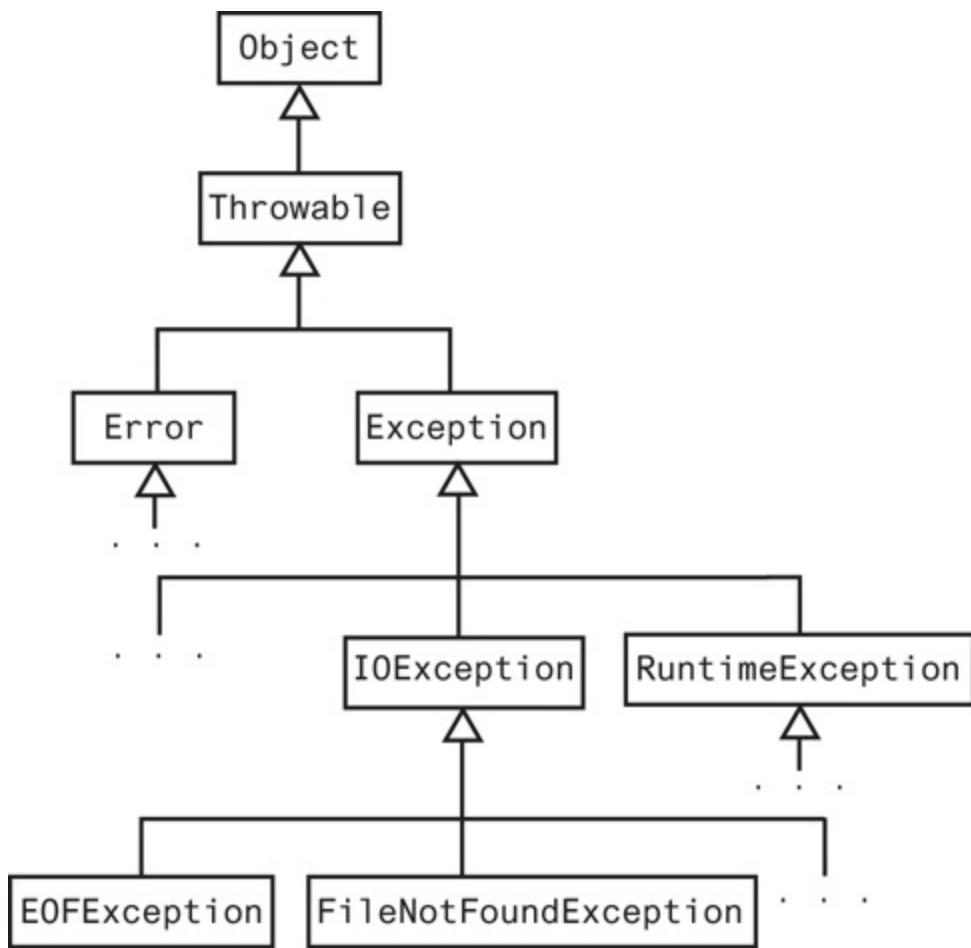
To detect that an exception has been thrown and prevent it from halting your application, Java allows you to create exception handlers. An *exception handler* is a section of code that gracefully responds to exceptions when they are thrown. The process of intercepting and responding to exceptions is called *exception handling*. If your code does not handle an exception when it is thrown, the *default exception handler* deals with it, as shown in [Code Listing 10-1](#). The default exception handler prints an error message and crashes the program.

The error that caused the exception to be thrown in [Code Listing 10-1](#) is easy to avoid. If the loop was written properly, it would not have tried to read outside the bounds of the array. Some errors, however, are caused by conditions that are outside the application and cannot be avoided. For example, suppose an application creates a file on the disk and the user deletes it. Later, the application attempts to open the file to read from it, and because it does not exist, an error occurs. As a result, an exception is thrown.

# Exception Classes

As previously mentioned, an exception is an object. Exception objects are created from classes in the Java API. The API has an extensive hierarchy of exception classes. A small part of the hierarchy is shown in [Figure 10-1](#).

# Figure 10-1 Part of the exception class hierarchy



[Figure 10-1 Full Alternative Text](#)

As you can see, all of the classes in the hierarchy inherit from the `Throwable` class. Just below the `Throwable` class are the classes `Error` and `Exception`. The `Error` class is intended to be a superclass for exceptions that are thrown when a critical error occurs, such as an internal error in the Java Virtual Machine, or running out of memory. Your applications should not try to handle these errors because they are the result of a serious condition.

All of the exceptions that you will handle are instances of classes that inherit

from `Exception`. [Figure 10-1](#) shows two of these classes: `IOException` and `RuntimeException`. These classes also serve as superclasses. `IOException` serves as a superclass for exceptions that are related to input and output operations. `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript.

The chart in [Figure 10-1](#) shows two of the classes that inherit from the `IOException` class: `EOFException` and `FileNotFoundException`. These are examples of classes that exception objects are created from. An `EOFException` object is thrown when an application attempts to read beyond the end of a file, and a `FileNotFoundException` object is thrown when an application tries to open a file that does not exist.



## Note:

The exception classes are in packages in the Java API. For example, `FileNotFoundException` is in the `java.io` package. When you handle an exception that is not in the `java.lang` package, you will need the appropriate `import` statement.

# Handling an Exception

To handle an exception, you use a `try` statement. We will look at several variations of the `try` statement, beginning with the following general format:

```
try
{
 (try block statements . . .)
}
catch (ExceptionType ParameterName)
{
 (catch block statements . . .)
}
```

First, the key word `try` appears. Next, a block of code appears inside braces, which are required. This block of code is known as a *try block*. A *try block* is

one or more statements that are executed and can potentially throw an exception. You can think of the code in the try block as being “protected” because the application will not halt if the try block throws an exception.

After the try block, a catch clause appears. A catch clause begins with the key word `catch`, followed by the code (*ExceptionType ParameterName*). This is a parameter variable declaration, where *ExceptionType* is the name of an exception class, and *ParameterName* is a variable name. If code in the try block throws an exception of the *ExceptionType* class, then the parameter variable will reference the exception object. In addition, the code that immediately follows the catch clause is executed. The code that immediately follows the catch clause is known as a *catch block*. Once again, the braces are required.

Let’s look at an example of code that uses a `try` statement. The statement inside the following try block attempts to open the file *MyFile.txt*. If the file does not exist, the `Scanner` object throws an exception of the `FileNotFoundException` class. This code is designed to handle that exception if it is thrown.

```
try
{
 File file = new File("MyFile.txt");
 Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
 System.out.println("File not found.");
}
```

Let’s look closer. First, the code in the try block is executed. If this code throws an exception, the Java Virtual Machine searches for a catch clause that can deal with the exception. In order for a catch clause to be able to deal with an exception, its parameter must be of a type that is compatible with the exception’s type. Here is this code’s catch clause:

```
catch (FileNotFoundException e)
```

This catch clause declares a reference variable named `e` as its parameter. The `e` variable can reference an object of the `FileNotFoundException` class. So,

this catch clause can deal with an exception of the `FileNotFoundException` class. If the code in the try block throws an exception of the `FileNotFoundException` class, the `e` variable will reference the exception object, and the code in the catch block will execute. In this case, the message “File not found.” will be printed. After the catch block is executed, the program will resume with the code that appears after the entire try/catch construct.



## Note:

The Java API documentation lists all of the exceptions that can be thrown from each method.

[Code Listing 10-2](#) shows a program that asks the user to enter a file name, then opens the file. If the file does not exist, an error message is printed.

## Code Listing 10-2 (`OpenFile.java`)

```
1 import java.io.*; // For File class and FileNotFoundException
2 import java.util.Scanner; // For the Scanner class
3
4 /**
5 * This program demonstrates how a FileNotFoundException
6 * exception can be handled.
7 */
8
9 public class OpenFile
10 {
11 public static void main(String[] args)
12 {
13 // Create a Scanner object for keyboard input.
14 Scanner keyboard = new Scanner(System.in);
15
16 // Get a file name from the user.
17 System.out.print("Enter the name of a file: ");
18 String fileName = keyboard.nextLine();
19
20 // Attempt to open the file.
```

```
21 try
22 {
23 // Create a File object representing the file.
24 File file = new File(fileName);
25
26 // Create a Scanner object to read the file.
27 // If the file does not exist, the following
28 // statement will throw a FileNotFoundException.
29 Scanner inputFile = new Scanner(file);
30
31 // If the file was successfully opened, the
32 // following statement will execute.
33 System.out.println("The file was found.");
34 }
35 catch (FileNotFoundException e)
36 {
37 // If the file was not found, the following
38 // statement will execute.
39 System.out.println("File not found.");
40 }
41
42 System.out.println("Done.");
43 }
44 }
```

## Program Output with Example Input Shown in Bold

(Assume *BadFile.txt* does not exist.)

Enter the name of a file: **BadFile.txt**   
File not found.  
Done.

## Program Output with Example Input Shown in Bold

(Assume *GoodFile.txt* does exist.)

Enter the name of a file: **GoodFile.txt**   
The file was found.  
Done.

Look at the first example run of the program. The user entered *BadFile.txt* as the file name. In line 24, inside the try block, a `File` object is created and this name is passed to the `File` constructor. In line 29, a reference to the `File` object is passed to the `Scanner` constructor. Because *BadFile.txt* does not

exist, an exception of the `FileNotFoundException` class is thrown by the `Scanner` class constructor. When the exception is thrown, the program immediately exits the try block, skipping the remaining lines in the block (lines 30 through 33). The program jumps to the catch clause in line 35, which has a `FileNotFoundException` parameter, and executes the catch block that follows it. [Figure 10-2](#) illustrates this sequence of events.

## Figure 10-2 Sequence of events with an exception

```
try
{
 // Create a File object representing the file.
 File file = new File(fileName);

 // Create a Scanner object to read the file.
 // If the file does not exist, the following
 // statement will throw a FileNotFoundException.
 Scanner inputFile = new Scanner(file);

 ... then these lines are → { // If the file was successfully opened, the
 // following statement will execute.
 System.out.println("The file was found.");
 }

 catch (FileNotFoundException e)
 {
 // If the file was not found, the following
 // statement will execute.
 System.out.println("File not found.");
 }
}
```

If this statement throws an exception... →

If the exception is an object of the `FileNotFoundException` class, the program jumps to this catch clause.

[Figure 10-2 Full Alternative Text](#)

Notice that after the catch block executes, the program resumes at the statement that immediately follows the `try/catch` construct. This statement prints the message “Done.”

Now look at the second example run of the program. In this case, the user entered *GoodFile.txt*, which is the name of a file that exists. No exception was thrown in the try block, so the program skips the catch clause and its catch block and jumps directly to the statement that follows the try/catch construct. This statement prints the message “Done.” [Figure 10-3](#) illustrates this sequence of events.

## Figure 10-3 Sequence of events with no exception

```
try
{
 // Create a File object representing the file.
 File file = new File(fileName);

 // Create a Scanner object to read the file.
 // If the file does not exist, the following
 // statement will throw a FileNotFoundException.
 Scanner inputFile = new Scanner(file);

 // If the file was successfully opened, the
 // following statement will execute.
 System.out.println("The file was found.");
}

catch (FileNotFoundException e)
{
 // If the file was not found, the following
 // statement will execute.
 System.out.println("File not found.");
}

System.out.println("Done.");
```

If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.

[Figure 10-3 Full Alternative Text](#)

# Retrieving the Default Error Message

Each exception object has a method named `getMessage` that can be used to retrieve the default error message for the exception. This is the same message that is displayed when the exception is not handled and the application halts. The program in [Code Listing 10-3](#) demonstrates the `getMessage` method. This is a modified version of the program in [Code Listing 10-2](#).

## Code Listing 10-3 (ExceptionMessage.java)

```
1 import java.io.*; // For File class and FileNotFoundException
2 import java.util.Scanner; // For the Scanner class
3
4 /**
5 * This program demonstrates how the default error message
6 * can be retrieved from an exception object.
7 */
8
9 public class ExceptionMessage
10 {
11 public static void main(String[] args)
12 {
13 // Create a Scanner object for keyboard input.
14 Scanner keyboard = new Scanner(System.in);
15
16 // Get a file name from the user.
17 System.out.print("Enter the name of a file: ");
18 String fileName = keyboard.nextLine();
19
20 // Attempt to open the file.
21 try
22 {
23 // Create a File object representing the file.
24 File file = new File(fileName);
25
26 // Create a Scanner object to read the file.
27 // If the file does not exist, the following
```

```

28 // statement will throw a FileNotFoundException.
29 Scanner inputFile = new Scanner(file);
30
31 // If the file was successfully opened, the
32 // following statement will execute.
33 System.out.println("The file was found.");
34 }
35 catch (FileNotFoundException e)
36 {
37 // If the file was not found, the following
38 // statement will execute. It displays the
39 // default error message.
40 System.out.println(e.getMessage());
41 }
42
43 System.out.println("Done.");
44 }
45 }
```

## Program Output with Example Input Shown in Bold

*(Assume BadFile.txt does not exist.)*

Enter the name of a file: **BadFile.txt**   
 BadFile.txt (The system cannot find the file specified)  
 Done.

[Code Listing 10-4](#) shows another example. This program forces the parseInt method of the Integer wrapper class to throw an exception.

## Code Listing 10-4 (ParseIntError.java)

```

1 /**
2 * This program demonstrates how the Integer.parseInt
3 * method throws an exception.
4 */
5
6 public class ParseIntError
7 {
8 public static void main(String[] args)
9 {
10 String str = "abcde";
```

```
11 int number;
12
13 try
14 {
15 // Try to convert str to an int.
16 number = Integer.parseInt(str);
17 }
18 catch (NumberFormatException e)
19 {
20 System.out.println("Conversion error: " +
21 e.getMessage());
22 }
23 }
24 }
```

## Program Output

Conversion error: For input string: "abcde"

The numeric wrapper classes’ “parse” methods all throw an exception of the NumberFormatException type if the string being converted does not contain a convertible numeric value. As you can see from the program, the exception’s getMessage method returns a string containing the value that could not be converted.

# Polymorphic References to Exceptions

Recall from [Chapter 9](#), a reference variable of a superclass type can reference objects that inherit from that superclass. This is called polymorphism. When handling exceptions, you can use a polymorphic reference as a parameter in the catch clause. For example, all of the exceptions that we have dealt with inherit from the Exception class. So, a catch clause that uses a parameter variable of the Exception type is capable of catching any exception that inherits from the Exception class. For example, the try statement in [Code Listing 10-4](#) could be written as follows:

```
try
```

```

{
 // Try to convert str to an int.
 number = Integer.parseInt(str);
}
catch (Exception e)
{
 System.out.println("The following error occurred: " +
 e.getMessage());
}

```

Although the `Integer` class's `parseInt` method throws a `NumberFormatException` object, this code still works because the `NumberFormatException` class inherits from the `Exception` class.

## Handling Multiple Exceptions

The programs we have studied so far test only for a single type of exception. In many cases, however, the code in the try block will be capable of throwing more than one type of exception. In such a case, you need to write a catch clause for each type of exception that could potentially be thrown.

For example, the program in [Code Listing 10-5](#) reads the contents of a file named

*SalesData.txt*. Each line in the file contains the sales amount for one month, and the file has several lines. Here are the contents of the file:

```

24987.62
26978.97
32589.45
31978.47
22781.76
29871.44

```

The program in [Code Listing 10-5](#) reads each number from the file and adds it to an accumulator variable. The try block contains code that can throw different types of exceptions. For example, the `Scanner` class's constructor can throw a `FileNotFoundException` if the file is not found, and the `Scanner` class's `nextDouble` method can throw an `InputMismatchException` (which is in the `java.util` package) if it reads a nonnumeric value from the file. To

handle these exceptions, the try statement has two catch clauses.

## Code Listing 10-5 (SalesReport.java)

```
1 import java.io.*; // For File class and FileNotFoundException
2 import java.util.*; // For Scanner and InputMismatchException
3
4 /**
5 * This program demonstrates how multiple exceptions can
6 * be caught with one try statement.
7 */
8
9 public class SalesReport
10 {
11 public static void main(String[] args)
12 {
13 String filename = "SalesData.txt"; // File name
14 int months = 0; // Month counter
15 double oneMonth; // One month's sales
16 double totalSales = 0.0; // Total sales
17 double averageSales; // Average sales
18
19 try
20 {
21 // Open the file.
22 File file = new File(filename);
23 Scanner inputFile = new Scanner(file);
24
25 // Process the contents of the file.
26 while (inputFile.hasNext())
27 {
28 // Get a month's sales amount.
29 oneMonth = inputFile.nextDouble();
30
31 // Accumulate the amount.
32 totalSales += oneMonth;
33
34 // Increment the month counter
35 months++;
36 }
37
38 // Close the file.
```

```

39 inputFile.close();
40
41 // Calculate the average.
42 averageSales = totalSales / months;
43
44 // Display the results.
45 System.out.printf("Number of months: %s\n", months);
46 System.out.printf("Total Sales: $%,.2f\n", totalSales);
47 System.out.printf("Average Sales: $%,.2f\n", averageSales
48 }
49 catch(FileNotFoundException e)
50 {
51 // The file was not found.
52 System.out.println("The file " + filename +
53 " does not exist.");
54 }
55 catch(InputMismatchException e)
56 {
57 // Thrown by the Scanner class's nextDouble
58 // method when a nonnumeric value is found.
59 System.out.println("Nonnumeric data " +
60 "found in the file:" +
61 e.getMessage());
62 }
63 }
64 }
```

## **Program Output**

```

Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95
```

When an exception is thrown by code in the try block, the JVM begins searching the try statement for a catch clause that can handle it. It searches the catch clauses from top to bottom, and passes control of the program to the first catch clause with a parameter that is compatible with the exception.

# **Using Exception Handlers to Recover from Errors**

The program in [Code Listing 10-5](#) demonstrates how a try statement can have several catch clauses in order to handle different types of exceptions. However, the program does not use the exception handlers to recover from any of the errors. Regardless of whether the file is not found or a nonnumeric item is encountered in the file, this program still halts. The program in [Code Listing 10-6](#) is a better example of effective exception handling. It attempts to recover from as many of the exceptions as possible.

## Code Listing 10-6 (SalesReport2.java)

```
1 import java.io.*; // For File class and FileNotFoundException
2 import java.util.*; // For Scanner and InputMismatchException
3
4 /**
5 * This program demonstrates how exception handlers can
6 * be used to recover from errors.
7 */
8
9 public class SalesReport2
10 {
11 public static void main(String[] args)
12 {
13 String filename = "SalesData.txt"; // File name
14 int months = 0; // Month counter
15 double oneMonth; // One month's sales
16 double totalSales = 0.0; // Total sales
17 double averageSales; // Average sales
18
19 // Attempt to open the file by calling the
20 // openfile method.
21 Scanner inputFile = openFile(filename);
22
23 // If the openFile method returned null, then
24 // the file was not found. Get a new file name.
25 while (inputFile == null)
26 {
27 Scanner keyboard = new Scanner(System.in);
28 System.out.print("ERROR: " + filename +
29 " does not exist.\n" +
30 "Enter another file name: ");
31 filename = keyboard.nextLine();
```

```
32 inputFile = openFile(filename);
33 }
34
35 // Process the contents of the file.
36 while (inputFile.hasNext())
37 {
38 try
39 {
40 // Get a month's sales amount.
41 oneMonth = inputFile.nextDouble();
42
43 // Accumulate the amount.
44 totalSales += oneMonth;
45
46 // Increment the month counter.
47 months++;
48 }
49 catch(InputMismatchException e)
50 {
51 // Display an error message.
52 // Nonnumeric data was encountered.
53 System.out.println("Nonnumeric data " +
54 "encountered in the file: " +
55 e.getMessage());
56
57 System.out.println("The invalid record " +
58 "will be skipped.");
59
60 // Skip past the invalid data.
61 inputFile.nextLine();
62 }
63 }
64
65 // Close the file.
66 inputFile.close();
67
68 // Calculate the average.
69 averageSales = totalSales / months;
70
71 // Display the results.
72 System.out.printf("Number of months: %s\n", months);
73 System.out.printf("Total Sales: $%,.2f\n", totalSales);
74 System.out.printf("Average Sales: $%,.2f\n", averageSales)
75 }
76
77 /**
78 * The openFile method opens the file with the name specificie
79 * by the argument. A reference to a Scanner object is
```

```
80 * returned.
81 */
82
83 public static Scanner openFile(String filename)
84 {
85 Scanner scan;
86
87 // Attempt to open the file.
88 try
89 {
90 File file = new File(filename);
91 scan = new Scanner(file);
92 }
93 catch(FileNotFoundException e)
94 {
95 scan = null;
96 }
97
98 return scan;
99 }
100 }
```

Let's look at how this program recovers from a `FileNotFoundException`. The `openFile` method, in lines 83 through 99, accepts a file name as its argument. The method creates a `File` object (passing the file name to the constructor) and a `Scanner` object. If the `Scanner` class constructor throws a `FileNotFoundException`, the method returns `null`. Otherwise, it returns a reference to the `Scanner` object. In the `main` method, a loop is used (in lines 25 through 33) to ask the user for a different file name in the event that the `openFile` method returns `null`.

Now, let's look at how the program recovers from unexpectedly encountering a nonnumeric item in the file. The statement in line 41, which calls the `Scanner` class's `nextDouble` method, is wrapped in a `try` statement that catches the `InputMismatchException`. If this exception is thrown by the `nextDouble` method, the catch block in lines 49 through 62 displays a message indicating a nonnumeric item was encountered, and the invalid record will be skipped. The invalid data is then read from the file with the `nextLine` method in line 61. Because the statement `months++` in line 47 is in the `try` block, it will not be executed when the exception occurs, so the number of months will still be correct. The loop continues processing with the next line in the file.

Let's look at some examples of how the program recovers from these errors. Suppose we rename the *SalesData.txt* file to *SalesInfo.txt*. Here is an example running of the program:

### **Program Output with Example Input Shown in Bold**

```
ERROR: SalesData.txt does not exist.
Enter another file name: SalesInfo.txt
Number of months: 6
Total Sales: $169,187.71
Average Sales: $28,197.95
```

Now, suppose we change the name of the file back to *SalesData.txt* and edit its contents as follows:

```
24987.62
26978.97
abc
31978.47
22781.76
29871.44
```

Notice that the third item is no longer a number. Here is the output of the program:

### **Program Output**

```
Nonnumeric data encountered in the file: For input string: "abc"
The invalid record will be skipped.
Number of months: 5
Total Sales: $136,598.26
Average Sales: $27,319.65
```

## **Handle Each Exception Only Once in a try Statement**

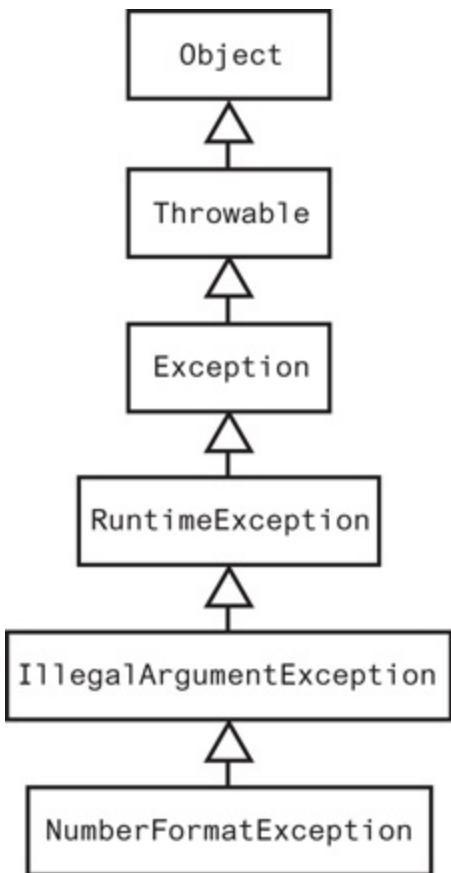
Not including polymorphic references, a **try** statement can have only one catch clause for each specific type of exception. For example, the following **try** statement will cause the compiler to issue an error message, because it

handles a `NumberFormatException` object with two catch clauses:

```
try
{
 number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
 System.out.println("Bad number format.");
}
// ERROR!!! NumberFormatException has already been caught!
catch (NumberFormatException e)
{
 System.out.println(str + " is not a number.");
}
```

Sometimes you can cause this error by using polymorphic references. For example, look at [Figure 10-4](#), which shows an inheritance hierarchy for the `NumberFormatException` class.

## Figure 10-4 Inheritance hierarchy for the `NumberFormatException` class



[Figure 10-4 Full Alternative Text](#)

As you can see from the figure, the `NumberFormatException` class inherits from the `IllegalArgumentException` class. Now look at the following code:

```
try
{
 number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
 System.out.println("Bad number format.");
}
// This will also cause an error.
catch (NumberFormatException e)
{
 System.out.println(str + " is not a number.");
}
```

The compiler issues an error message regarding the second catch clause, reporting that `NumberFormatException` has already been caught. This is

because the first catch clause, which catches `IllegalArgumentException` objects, will polymorphically catch `NumberFormatException` objects.

When in the same try statement you are handling multiple exceptions and some of the exceptions are related to each other through inheritance, then you should handle the more specialized exception classes before the more general exception classes. We can rewrite the previous code as follows, with no errors:

```
try
{
 number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
 System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e)
{
 System.out.println("Bad number format.");
}
```

## The finally Clause

The try statement may have an optional `finally` clause, which must appear after all of the catch clauses. Here is the general format of a try statement with a `finally` clause:

```
try
{
 (try block statements . . .)
}
catch (ExceptionType ParameterName)
{
 (catch block statements . . .)
}
finally
{
 (finally block statements . . .)
}
```

The *finally block* is one or more statements that are always executed after the try block has executed, and after any catch blocks have executed if an exception was thrown. The statements in the finally block execute whether an exception occurs or not. For example, the following code opens a file of doubles and reads its contents. The outer try statement opens the file and has a catch clause that catches the `FileNotFoundException`. The inner try statement reads values from the file and has a catch clause that catches the `InputMismatchException`. The finally block closes the file, regardless of whether an `InputMismatchException` occurs.

```
try
{
 // Open the file.
 File file = new File(filename);
 Scanner inputFile = new Scanner(file);

 try
 {
 // Read and display the file's contents.
 while (inputFile.hasNext())
 {
 System.out.println(inputFile.nextDouble());
 }
 }
 catch (InputMismatchException e)
 {
 System.out.println("Invalid data found.");
 }
 finally
 {
 // Close the file.
 inputFile.close();
 }
}
catch (FileNotFoundException e)
{
 System.out.println("File not found.");
}
```

## The Stack Trace

Quite often, a method will call another method, which will call yet another

method. For example, method A calls method B, which calls method C. The *call stack* is an internal list of all the methods that are currently executing.

When an exception is thrown by a method that is executing under several layers of method calls, it is sometimes helpful to know which methods were responsible for the method being called. A *stack trace* is a list of all the methods in the call stack. It indicates the method that was executing when an exception occurred and all of the methods that were called in order to execute that method. For example, look at the program in [Code Listing 10-7](#). It has three methods: main, myMethod, and produceError. The main method calls myMethod, which calls produceError. The produceError method causes an exception by passing an invalid position number to the String class's charAt method. The exception is not handled by the program, but is dealt with by the default exception handler.

## Code Listing 10-7 (StackTrace.java)

```
1 /**
2 * This program demonstrates the stack trace that is
3 * produced when an exception is thrown.
4 */
5
6 public class StackTrace
7 {
8 public static void main(String[] args)
9 {
10 System.out.println("Calling myMethod...");
11 myMethod();
12 System.out.println("Method main is done.");
13 }
14
15 /**
16 * myMethod
17 */
18
19 public static void myMethod()
20 {
21 System.out.println("Calling produceError...");
22 produceError();
```

```
23 System.out.println("myMethod is done.");
24 }
25
26 /**
27 * produceError
28 */
29
30 public static void produceError()
31 {
32 String str = "abc";
33
34 // The following statement will cause an error.
35 System.out.println(str.charAt(3));
36 System.out.println("produceError is done.");
37 }
38 }
```

## Program Output

```
Calling myMethod...
Calling produceError...
Exception in thread "main" java.lang.StringIndexOutOfBoundsException
String index out of range: 3
at java.lang.String.charAt(String.java:687)
at StackTrace.produceError(StackTrace.java:35)
at StackTrace.myMethod(StackTrace.java:22)
at StackTrace.main(StackTrace.java:11)
```

When the exception occurs, the error message shows a stack trace listing the methods that were called in order to produce the exception. The first method that is listed, `charAt`, is the method that is responsible for the exception. The next method, `produceError`, is the method that called `charAt`. The next method, `myMethod`, is the method that called `produceError`. The last method, `main`, is the method that called `myMethod`. The stack trace shows the chain of methods that were called when the exception was thrown.



## Note:

All exception objects have a `printStackTrace` method, inherited from the `Throwable` class, that prints a stack trace.

# Handling Multiple Exceptions with One catch Clause

In versions of Java prior to Java 7, each catch clause can handle only one type of exception. Beginning in Java 7, however, a catch clause can handle more than one type of exception. This can reduce a lot of duplicated code in a try statement that needs to catch multiple exceptions, but performs the same operation for each one. For example, suppose we have the following try statement in a program:

```
try
{
 (try block statements...)
}
catch(NumberFormatException ex)
{
 respondToError();
}
catch(IOException ex)
{
 respondToError();
}
```

This try statement has two catch clauses: one that handles a NumberFormatException, and another that handles an IOException. Notice both catch blocks do the same thing: they call a method named respondToError. Since both catch blocks perform the same operation, the catch clauses can be combined into a single catch clause that handles both types of exception, as shown here:

```
try
{
 (try block statements...)
}
catch(NumberFormatException | IOException ex)
{
 respondToError();
}
```

Notice in the catch clause, the exception types are separated by a | symbol,

which is the same symbol used as the logical OR operator. You can think of this as meaning that the clause will catch a `NumberFormatException` or an `IOException`. The following code shows a catch clause that handles three types of exceptions:

```
try
{
 (try block statements...)
}
catch(NumberFormatException | IOException | InputMismatchException)
{
 respondToError();
}
```

In this code, the catch clause will handle a `NumberFormatException`, or an `IOException`, or an `InputMismatchException`.

The ability to catch multiple types of exceptions with a single catch clause is known as *multi-catch* and was introduced in Java 7. [Code Listing 10-8](#) shows a complete program that uses multi-catch. The catch clause in line 34 can handle a `FileNotFoundException` or an `InputMismatchException`.

## Code Listing 10-8 (`MultiCatch.java`)

```
1 import java.io.*; // For File class and FileNotFoundException
2 import java.util.*; // For Scanner and InputMismatchException
3
4 /**
5 * This program demonstrates how multiple exceptions can
6 * be caught with a single catch clause.
7 */
8
9 public class MultiCatch
10 {
11 public static void main(String[] args)
12 {
13 int number; // To hold a number from the file
14
15 try
16 {
```

```

17 // Open the file.
18 File file = new File("Numbers.txt");
19 Scanner inputFile = new Scanner(file);
20
21 // Process the contents of the file.
22 while (inputFile.hasNext())
23 {
24 // Get a number from the file.
25 number = inputFile.nextInt();
26
27 // Display the number.
28 System.out.println(number);
29 }
30
31 // Close the file.
32 inputFile.close();
33 }
34 catch(FileNotFoundException | InputMismatchException ex)
35 {
36 // Display an error message.
37 System.out.println("Error processing the file.");
38 }
39 }
40 }
```



## Note:

If you are using a version of Java prior to Java 7, you cannot use multi-catch.

# When an Exception Is Not Caught

When an exception is thrown, it cannot be ignored. It must be handled by the program, or by the default exception handler. When the code in a method throws an exception, the normal execution of that method stops, and the JVM searches for a compatible exception handler inside the method. If there is no code inside the method to handle the exception, then control of the program is passed to the previous method in the call stack (that is, the method that called the offending method). If that method cannot handle the exception, then control is passed again, up the call stack, to the previous method. This

continues until control reaches the `main` method. If the `main` method does not handle the exception, then the program is halted, and the default exception handler handles the exception.

This was the case for the program in [Code Listing 10-7](#). Because the `produceError` method did not handle the exception, control was passed back to `myMethod`. It didn't handle the exception either, so control was passed back to `main`. Because `main` didn't handle the exception, the program halted and the default exception handler displayed the error messages.

## Checked and Unchecked Exceptions

In Java, there are two categories of exceptions: unchecked and checked. *Unchecked exceptions* are those that inherit from the `Error` class or the `RuntimeException` class. Recall that the exceptions that inherit from `Error` are thrown when a critical error occurs, such as running out of memory. You should not handle these exceptions, because the conditions that cause them can rarely be dealt with in the program. Also recall that `RuntimeException` serves as a superclass for exceptions that result from programming errors, such as an out-of-bounds array subscript. It is best not to handle these exceptions either, because they can be avoided with properly written code. So, you should not handle unchecked exceptions.

All of the remaining exceptions (that is, those that do *not* inherit from `Error` or `RuntimeException`) are *checked exceptions*. These are the exceptions you should handle in your program. If the code in a method can potentially throw a checked exception, then that method must meet one of the following requirements:

- It must handle the exception, or;
- It must have a `throws` clause listed in the method header.

The `throws` clause informs the compiler of the exceptions that could get thrown from a method. For example, look at the following method:

```
// This method will not compile!
```

```

public void displayFile(String name)
{
 // Open the file.
 File file = new File(name);
 Scanner inputFile = new Scanner(file);

 // Read and display the file's contents.
 while (inputFile.hasNext())
 {
 System.out.println(inputFile.nextLine());
 }

 // Close the file.
 inputFile.close();
}

```

The code in this method is capable of throwing a `FileNotFoundException`, which is a checked exception. Because the method does not handle this exception, it must have a `throws` clause in its header, or it will not compile.

The key word `throws` is written at the end of the method header, followed by a list of the types of exceptions that the method can throw. Here is the revised method header:

```
public void displayFile(String name) throws FileNotFoundException
```

The `throws` clause tells the compiler that this method can throw a `FileNotFoundException`. (If there is more than one type of exception, you separate them with commas.)

Now you know why you wrote a `throws` clause on any method that performed file operations in the previous chapters. We did not handle any of the checked exceptions that `Scanner` objects can throw, so we had to inform the compiler that our methods might pass them up the call stack.



## Checkpoint

1. 10.1 Briefly describe what an exception is.
2. 10.2 What does it mean to “throw” an exception?

3. 10.3 If an exception is thrown and the program does not handle it, what happens?
4. 10.4 Other than the `Object` class, from what class do all exceptions inherit?
5. 10.5 What is the difference between exceptions that inherit from the `Error` class, and exceptions that inherit from the `Exception` class?
6. 10.6 What is the difference between a try block and a catch block?
7. 10.7 After the catch block has handled the exception, where does program execution resume?
8. 10.8 How do you retrieve an error message from an exception?
9. 10.9 If multiple exceptions can be thrown by code in a try block, how does the JVM know to which catch clause it should pass the control of the program?
10. 10.10 When does the code in a finally block execute?
11. 10.11 What is the call stack? What is a stack trace?
12. 10.12 A program's `main` method calls method A, which calls method B. None of these methods perform any exception handling. The code in method B throws an exception. Describe what happens.
13. 10.13 What are the differences between a checked and an unchecked exception?
14. 10.14 When are you required to have a `throws` clause in a method header?

# 10.2 Throwing Exceptions

## Concept:

You can write code that throws one of the standard Java exceptions, or an instance of a custom exception class that you have designed.

You can use the `throw` statement to manually throw an exception. The general format of the `throw` statement is:

```
throw new ExceptionType(MessageString);
```

The `throw` statement causes an exception object to be created and thrown. In this general format, `ExceptionType` is an exception class name, and `MessageString` is an optional `String` argument passed to the exception object's constructor. The `MessageString` argument contains a custom error message that can be retrieved from the exception object's `getMessage` method. If you do not pass a message to the constructor, the exception will have a `null` message. Here is an example of a `throw` statement:

```
throw new Exception("Out of fuel");
```

This statement creates an object of the `Exception` class and passes the string “Out of fuel” to the object's constructor. The object is then thrown, which causes the exception-handling process to begin.



## Note:

Don't confuse the `throw` statement with the `throws` clause. The `throw` statement causes an exception to be thrown. The `throws` clause informs the compiler that a method throws one or more exceptions.

Recall the `InventoryItem` class from [Chapter 6](#). This class holds simple data about an item in an inventory. A description of the item is stored in the `description` field, and the number of units on hand is stored in the `units` field. [Figure 10-5](#) shows a UML diagram for the class.

## Figure 10-5 UML diagram for the `InventoryItem` class

| InventoryItem                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| - <code>description</code> : String<br>- <code>units</code> : int                                                                                                                                                                                                                                             |
| + <code>InventoryItem()</code><br>+ <code>InventoryItem(d : String)</code><br>+ <code>InventoryItem(d : String, u : int)</code><br>+ <code>setDescription(d : String) : void</code><br>+ <code>setUnits(u : int) : void</code><br>+ <code>getDescription() : String</code><br>+ <code>getUnits() : int</code> |

[Figure 10-5 Full Alternative Text](#)

The second constructor accepts a `String` argument for the `description` field. The third constructor accepts a `String` argument for the `description` field, and an `int` argument for the `units` field. Suppose we want to prevent invalid data from being passed to the constructors. For example, we want to prevent an empty string from being passed into the `description` field, and a negative number from being passed into the `units` field. One way to accomplish this is to have the constructors throw an exception when invalid data is passed as arguments.

Here is the code for the second constructor, written to throw an exception when an empty string is passed as the argument:

```
public InventoryItem(String d)
{
 if (d.equals(""))
 {
 throw new IllegalArgumentException("Description is an empty st
 }
 description = d;
 units = 0;
}
```

This constructor throws an `IllegalArgumentException` if the `d` parameter contains an empty string. The message “Description is an empty string” is passed to the exception object’s constructor. When we catch this exception, we can retrieve the message by calling the object’s `getMessage` method. The `IllegalArgumentException` class was chosen for this error condition because it seems like the most appropriate exception to throw in response to an illegal argument being passed to the constructor.

(`IllegalArgumentException` inherits from `RuntimeException`, which inherits from `Exception`.)

Here is the code for the third constructor, written to throw an exception when an empty string is passed as the `d` parameter, or a negative number is passed into the `u` parameter:

```
public InventoryItem(String d, int u)
{
 if (d.equals(""))
 {
 throw new IllegalArgumentException("Description is an empty s
 }
 if (u < 0)
 throw new IllegalArgumentException("Units is negative.");
 description = d;
 units = u;
}
```



## Note:

Because the `IllegalArgumentException` class inherits from the `RuntimeException` class, it is unchecked. If we had chosen a checked

exception class, we would have to put a `throws` clause in each of these constructor's headers.

The program in [Code Listing 10-9](#) demonstrates how these constructors work.

## Code Listing 10-9 (InventoryDemo.java)

```
1 /**
2 * This program demonstrates how the InventoryItem class
3 * throws exceptions.
4 */
5
6 public class InventoryDemo
7 {
8 public static void main(String[] args)
9 {
10 InventoryItem item;
11
12 // Try to assign an empty string to the
13 // description field.
14 try
15 {
16 item = new InventoryItem("");
17 }
18 catch (IllegalArgumentException e)
19 {
20 System.out.println(e.getMessage());
21 }
22
23 // Again, try to assign an empty string to
24 // the description field.
25 try
26 {
27 item = new InventoryItem("", 5);
28 }
29 catch (IllegalArgumentException e)
30 {
31 System.out.println(e.getMessage());
32 }
33
34 // Try to assign a negative number to the
```

```
35 // units field.
36 try
37 {
38 item = new InventoryItem("Wrench", -1);
39 }
40 catch (IllegalArgumentException e)
41 {
42 System.out.println(e.getMessage());
43 }
44 }
45 }
```

## Program Output

```
Description is an empty string.
Description is an empty string.
Units is negative.
```

# Creating Your Own Exception Classes

To meet the needs of a specific class or application, you can create your own exception classes by extending the `Exception` class or one of its subclasses.

Let's look at an example that uses programmer-defined exceptions. Recall the `BankAccount` class from [Chapter 3](#). This class holds the data for a bank account. A UML diagram for the class is shown in [Figure 10-6](#).

**Figure 10-6 UML diagram for the `BankAccount` class**

| BankAccount                                                                                                                                                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- balance : double</li> <li>- interestRate : double</li> <li>- interest : double</li> </ul>                                                                                                                                                                                       |
| <ul style="list-style-type: none"> <li>+ BankAccount(startBalance : double,<br/>                  intRate : double)</li> <li>+ deposit(amount : double) : void</li> <li>+ withdraw(amount : double) : void</li> <li>+ addInterest() : void</li> <li>+ getBalance() : double</li> <li>+ getInterest() : double</li> </ul> |

[Figure 10-6 Full Alternative Text](#)

There are a number of errors that could cause a `BankAccount` object to incorrectly perform its duties. Here are some specific examples:

- A negative starting balance is passed to the constructor.
- A negative interest rate is passed to the constructor.
- A negative number is passed to the `deposit` method.
- A negative number is passed to the `withdraw` method.
- The amount passed to the `withdraw` method exceeds the account's balance.

We can create our own exceptions that represent each of these error conditions. Then we can rewrite the class so it throws one of our custom exceptions when any of these errors occur. Let's start by creating an exception class for a negative starting balance. [Code Listing 10-10](#) shows an exception class named `NegativeStartingBalance`.

# Code Listing 10-10

## (NegativeStartingBalance.java)

```
1 /**
2 * NegativeStartingBalance exceptions are thrown by
3 * the BankAccount class when a negative starting
4 * balance is passed to the constructor.
5 */
6
7 public class NegativeStartingBalance extends Exception
8 {
9 /**
10 * No-arg constructor
11 */
12
13 public NegativeStartingBalance()
14 {
15 super("Error: Negative starting balance");
16 }
17
18 /**
19 * The following constructor accepts the amount
20 * that was given as the starting balance.
21 */
22
23 public NegativeStartingBalance(double amount)
24 {
25 super("Error: Negative starting balance: " +
26 amount);
27 }
28 }
```

Notice this class inherits from the `Exception` class. It has two constructors. The no-arg constructor passes the string “Error: Negative starting balance” to the superclass constructor. This is the error message that is retrievable from an object’s `getMessage` method. The second constructor accepts the starting balance as a double argument. This amount is used to pass a more detailed error message containing the starting balance amount to the superclass.

A similar class can be written to handle negative interest rates. [Code Listing 10-11](#) shows the `NegativeInterestRate` class. This class is also derived from the `Exception` class.

# Code Listing 10-11

## (NegativeInterestRate.java)

```
1 /**
2 * NegativeInterestRate exceptions are thrown by the
3 * BankAccount class when a negative interest rate is
4 * passed to the constructor.
5 */
6
7 public class NegativeInterestRate extends Exception
8 {
9 /**
10 * No-arg constructor
11 */
12
13 public NegativeInterestRate()
14 {
15 super("Error: Negative interest rate");
16 }
17
18 /**
19 * The following constructor accepts the amount that
20 * was given as the interest rate.
21 */
22
23 public NegativeInterestRate(double amount)
24 {
25 super("Error: Negative interest rate: " + amount);
26 }
27 }
```

The BankAccount constructor can now be rewritten, as follows, to throw a NegativeStartingBalance exception when a negative value is passed as the starting balance, or a NegativeInterestRate exception when a negative number is passed as the interest rate:

```
public BankAccount(double startBalance,
 double intRate) throws NegativeStartingBalance,
 NegativeInterestRate
{
 if (startBalance < 0)
 throw new NegativeStartingBalance(startBalance);
 if (intRate < 0)
```

```
 throw new NegativeInterestRate(intRate);

 balance = startBalance;
 interestRate = intRate;
 interest = 0.0;
}
```

Note that both NegativeStartingBalance and NegativeInterestRate inherit from the Exception class. This means both classes are checked exception classes. Because of this, the constructor header must have a throws clause listing these exception types.

The program in [Code Listing 10-12](#) demonstrates this constructor by forcing it to throw the exceptions.

## Code Listing 10-12 (AccountTest.java)

```
1 /**
2 * This program demonstrates how the BankAccount
3 * class constructor throws custom exceptions.
4 */
5
6 public class AccountTest
7 {
8 public static void main(String[] args)
9 {
10 // Force a NegativeStartingBalance exception.
11 try
12 {
13 BankAccount account = new BankAccount(-1, 0.04);
14 }
15 catch(NegativeStartingBalance e)
16 {
17 System.out.println(e.getMessage());
18 }
19 catch(NegativeInterestRate e)
20 {
21 System.out.println(e.getMessage());
22 }
23
24 // Force a NegativeInterestRate exception.
25 }
```

```
25 try
26 {
27 BankAccount account = new BankAccount(100, -0.04);
28 }
29 catch(NegativeStartingBalance e)
30 {
31 System.out.println(e.getMessage());
32 }
33 catch(NegativeInterestRate e)
34 {
35 System.out.println(e.getMessage());
36 }
37 }
38 }
```

## Program Output

```
Error: Negative starting balance: -1.0
Error: Negative interest rate: -0.04
```



## Checkpoint

1. 10.15 What does the `throw` statement do?
2. 10.16 What is the purpose of the argument that is passed to an exception object's constructor? What happens if you do not pass an argument to the constructor?
3. 10.17 What is the difference between the `throw` statement and the `throws` clause?
4. 10.18 If a method has a `throw` statement, does it always have to have a `throws` clause in its header? Why or why not?
5. 10.19 If you are writing a custom exception class, how can you make sure it is checked? How can you make sure it is unchecked?

# 10.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization

## Concept:

A file containing raw binary data is known as a binary file. The content of a binary file is not formatted as text, and not meant to be opened in a text editor. A random access file is a file that allows a program to read data from any location within the file, or write data to any location within the file. Object serialization is the process of converting an object to a series of bytes and saving them to a file. Deserialization is the process of reconstructing a serialized object.

## Binary Files

All the files with which you've been working so far have been text files. That means the data stored in the files has been formatted as text. Even a number, when stored in a text file with the `print` or `println` method, is converted to text. For example, consider the following program segment:

```
PrintWriter outputFile = new PrintWriter("Number.txt");
int x = 1297;
outputFile.print(x);
```

The last statement writes the contents of the variable `x` to the `Number.txt` file. When the number is written, however, it is stored as the characters '1', '2', '9', and '7'. This is illustrated in [Figure 10-7](#).

# **Figure 10-7 The number 1297 expressed as characters**

1297 expressed as characters.

|     |     |     |     |
|-----|-----|-----|-----|
| '1' | '2' | '9' | '7' |
|-----|-----|-----|-----|

When a number such as 1297 is stored in the computer's memory, it isn't stored as text. It is formatted as a binary number. [Figure 10-8](#) shows how the number 1297 is stored in memory, in an `int` variable, using binary. Recall `int` variables occupy four bytes.

# **Figure 10-8 The number 1297 as a binary number, as it is stored in memory**

1297 as a binary number.

|          |          |          |          |
|----------|----------|----------|----------|
| 00000000 | 00000000 | 00000101 | 00010001 |
|----------|----------|----------|----------|

The binary representation of the number shown in [Figure 10-8](#) is the way the raw data is stored in memory. In fact, this is sometimes called the *raw binary format*. Data can be stored in a file in its raw binary format. A file containing binary data is often called a *binary file*.

Storing data in its binary format is more efficient than storing it as text, because there are fewer conversions to take place. In addition, there are some types of data that should be stored only in its raw binary format. Images are an example. However, when data is stored in a binary file, you cannot open the file in a text editor such as Notepad. When a text editor opens a file, it assumes the file contains text.

# Writing Data to a Binary File

To write data to a binary file, you must create objects from the following classes:

This class allows you to open a file for writing binary `FileOutputStream` data and establish a connection with it. It provides only basic functionality for writing bytes to the file, however.

This class allows you to write data of any primitive type or `String` objects to a binary file. The `DataOutputStream` class by itself cannot directly access a file, however. It is used in conjunction with a `FileOutputStream` object that has a connection to a file.

You wrap a `DataOutputStream` object around a `FileOutputStream` object to write data to a binary file. The following code shows how a file named *MyInfo.dat* can be opened for binary output:

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat");
DataOutputStream outputFile = new DataOutputStream(fstream);
```

The first line creates an instance of the `FileOutputStream` class, which has the ability to open a file for binary output and establish a connection with it. You pass the name of the file that you wish to open, as a string, to the constructor. The second line creates an instance of the `DataOutputStream` object that is connected to the `FileOutputStream` referenced by `fstream`. The result of this statement is that the `outputFile` variable will reference an object that is able to write binary data to the *MyInfo.dat* file.



## Warning!

If the file that you are opening with the `FileOutputStream` object already exists, it will be deleted, and an empty file by the same name will be created.



## Note:

The `FileOutputStream` constructor throws an `IOException` if an error occurs when it attempts to open the file.

If there is no reason to reference the `FileOutputStream` object, these statements can be combined into one, as follows:

```
DataOutputStream outputFile =
 new DataOutputStream(new FileOutputStream("MyInfo.dat"));
```

Once the `DataOutputStream` object has been created, you can use it to write binary data to the file. [Table 10-1](#) lists some of the `DataOutputStream` methods. Note that each of the methods listed in the table throws an `IOException` if an error occurs.

## Table 10-1 Some of the DataOutputStream methods

| Method                                    | Description                                                                                                                                                          |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void close()</code>                 | Closes the file.                                                                                                                                                     |
| <code>void writeBoolean(boolean b)</code> | Writes the boolean value passed to <i>b</i> to the file.                                                                                                             |
| <code>void writeByte(byte b)</code>       | Writes the byte value passed to <i>b</i> to the file.                                                                                                                |
| <code>void writeChar(int c)</code>        | This method accepts an <code>int</code> which is assumed to be a character code. The character it represents is written to the file as a two-byte Unicode character. |
| <code>void writeDouble(double d)</code>   | Writes the <code>double</code> value passed to <i>d</i> to the file.                                                                                                 |
| <code>void</code>                         | Writes the <code>float</code> value passed to <i>f</i> to the                                                                                                        |

**writeFloat(float f)** file.  
**void writeInt(int i)** Writes the int value passed to *i* to the file.  
**void writeLong(long num)** Writes the long value passed to *num* to the file.  
**void writeShort(short s)** Writes the short value passed to *s* to the file.  
**void writeUTF(String str)** Writes the String object passed to *str* to the file using the Unicode Text Format.

The program in [Code Listing 10-13](#) shows a simple demonstration. An array of int values is written to the file *Numbers.dat*.

## Code Listing10-13 (**WriteBinaryFile.java**)

```

1 import java.io.*;
2
3 /**
4 * This program opens a binary file and writes the contents
5 * of an int array to the file.
6 */
7
8 public class WriteBinaryFile
9 {
10 public static void main(String[] args) throws IOException
11 {
12 // Create an array of integers.
13 int[] numbers = { 2, 4, 6, 8, 10, 12, 14 };
14
15 // Open a binary file for output.
16 FileOutputStream fstream =
17 new FileOutputStream("Numbers.dat");
18 DataOutputStream outputFile =
19 new DataOutputStream(fstream);
20
21 System.out.println("Writing to the file...");
22
23 // Write the array elements to the binary file.
24 for (int i = 0; i < numbers.length; i++)
25 outputFile.writeInt(numbers[i]);

```

```
26
27 // Close the file.
28 outputFile.close();
29 System.out.println("Done.");
30 }
31 }
```

## Program Output

```
Writing to the file...
Done.
```

# Reading Data from a Binary File

To open a binary file for input, you use the following classes:

`FileInputStream` This class allows you to open a file for reading binary data, and establish a connection with it. It provides only the basic functionality for reading bytes from the file, however.

`DataInputStream` This class allows you to read data of any primitive type, or `String` objects, from a binary file. The `DataInputStream` class by itself cannot directly access a file, however. It is used in conjunction with a `FileInputStream` object that has a connection to a file.

To open a binary file for input, you wrap a `DataInputStream` object around a `FileInputStream` object. The following code shows the file `MyInfo.dat` can be opened for binary input:

```
FileInputStream fstream = new FileInputStream("MyInfo.dat");
DataInputStream inputFile = new DataInputStream(fstream);
```

The following code, which combines these two statements into one, can also be used:

```
DataInputStream inputFile =
 new DataInputStream(new FileInputStream("MyInfo.dat"));
```

The `FileInputStream` constructor will throw a `FileNotFoundException` if the file named by the string argument cannot be found. Once the `DataInputStream` object has been created, you can use it to read binary data from the file. [Table 10-2](#) lists some of the `DataInputStream` methods. Note that each of the read methods listed in the table throws an `EOFException` if the end of the file has already been reached.

## Table 10-2 Some of the DataInputStream methods

| Method                             | Description                                                                                                                                                                                 |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void close()</code>          | Closes the file.                                                                                                                                                                            |
| <code>boolean readBoolean()</code> | Reads a boolean value from the file and returns it.                                                                                                                                         |
| <code>byte readByte()</code>       | Reads a byte value from the file and returns it.                                                                                                                                            |
|                                    | Reads a char value from the file and returns it.                                                                                                                                            |
| <code>char readChar()</code>       | The character is expected to be stored as a two-byte Unicode character, as written by the <code>DataOutputStream</code> class's <code>writeChar</code> method.                              |
| <code>double readDouble()</code>   | Reads a double value from the file and returns it.                                                                                                                                          |
| <code>float readFloat()</code>     | Reads a float value from the file and returns it.                                                                                                                                           |
| <code>int readInt()</code>         | Reads an int value from the file and returns it.                                                                                                                                            |
| <code>long readLong()</code>       | Reads a long value from the file and returns it.                                                                                                                                            |
| <code>short readShort()</code>     | Reads a short value from the file and returns it.                                                                                                                                           |
| <code>String readUTF()</code>      | Reads a string from the file and returns it as a <code>String</code> object. The string must have been written with the <code>DataOutputStream</code> class's <code>writeUTF</code> method. |

The program in [Code Listing 10-14](#) opens the `Numbers.dat` file created by the

program in [Code Listing 10-13](#). The numbers are read from the file then displayed on the screen. Notice the program must catch the `EOFException` in order to determine when the file's end has been reached.

## Code Listing 10-14 (`ReadBinaryFile.java`)

```
1 import java.io.*;
2
3 /**
4 * This program opens a binary file, then reads and displays
5 * the contents.
6 */
7
8 public class ReadBinaryFile
9 {
10 public static void main(String[] args) throws IOException
11 {
12 int number; // To hold a number
13 boolean endOfFile = false; // End of file flag
14
15 // Open Numbers.dat as a binary file.
16 FileInputStream fstream =
17 new FileInputStream("Numbers.dat");
18 DataInputStream inputFile =
19 new DataInputStream(fstream);
20
21 System.out.println("Reading numbers from the file:");
22
23 // Read data from the file.
24 while (!endOfFile)
25 {
26 try
27 {
28 number = inputFile.readInt();
29 System.out.print(number + " ");
30 }
31 catch (EOFException e)
32 {
33 endOfFile = true;
34 }
35 }
36 }
```

```
37 // Close the file.
38 inputFile.close();
39 System.out.println("\nDone.");
40 }
41 }
```

## Program Output

```
Reading numbers from the file:
2 4 6 8 10 12 14
Done.
```

# Writing and Reading Strings

To write a string to a binary file, you should use the `DataOutputStream` class's `writeUTF` method. This method writes its `String` argument in a format known as *UTF-8 encoding*. Here's how the encoding works: Just before writing the string, this method writes a two-byte integer indicating the number of bytes that the string occupies. Then, it writes the string's characters in Unicode. (UTF stands for Unicode Text Format.)

When the `DataInputStream` class's `readUTF` method reads from the file, it expects the first two bytes to contain the number of bytes that the string occupies. It then reads that many bytes, and returns them as a `String`.

For example, assuming that `outputFile` references a `DataOutputStream` object, the following code uses the `writeUTF` method to write a string:

```
String name = "Chloe";
outputFile.writeUTF(name);
```

Assuming that `inputFile` references a `DataInputStream` object, the following statement uses the `readUTF` method to read a UTF-8 encoded string from the file:

```
String name = inputFile.readUTF();
```

Remember that the `readUTF` method will correctly read a string only when the

string was written with the `writeUTF` method.



## Note:

The book's source code, available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis), contains the example programs `WriteUTF.java` and `ReadUTF.java`, which demonstrate writing and reading strings using these methods.

# Appending Data to an Existing Binary File

If you pass the name of an existing file to the `FileOutputStream` constructor, it will be deleted, and a new empty file with the same name will be created. Sometimes, however, you want to preserve an existing file and append new data to its current contents. The `FileOutputStream` constructor takes an optional second argument which must be a boolean value. If the argument is `true`, the file will not be deleted if it already exists, and new data will be written to the end of the file. If the argument is `false`, the file will be deleted if it already exists. For example, the following code opens the file `MyInfo.dat` for output. If the file exists, it will not be deleted, and any data written to the file will be appended to the existing data.

```
FileOutputStream fstream = new FileOutputStream("MyInfo.dat", true);
DataOutputStream outputFile = new DataOutputStream(fstream);
```

# Random Access Files

All of the programs you have created to access files so far have performed *sequential file access*. With sequential access, when a file is opened for input, its read position is at the very beginning of the file. This means that the first time data is read from the file, the data will be read from its beginning. As the reading continues, the file's read position advances sequentially through the

file's contents.

The problem with sequential file access is that in order to read a specific byte from the file, all the preceding bytes must be read first. For instance, if a program needs data stored at the hundredth byte of a file, it will have to read the first 99 bytes to reach it. If you've ever listened to a cassette tape player, you understand sequential access. To listen to a song at the end of the tape, you have to listen to all the songs that are before it, or fast-forward over them. There is no way to immediately jump to that particular song.

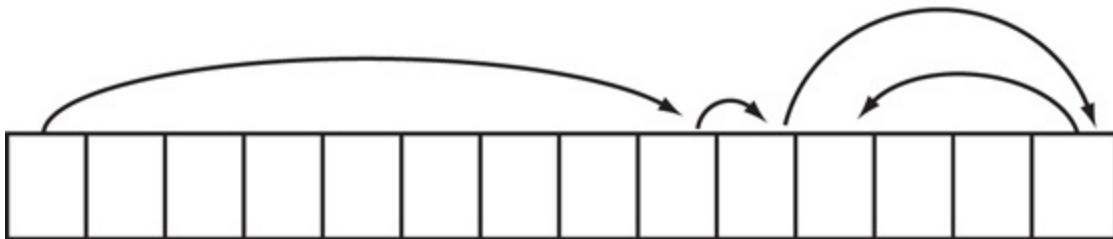
Although sequential file access is useful in many circumstances, it can slow a program down tremendously. If the file is very large, locating data buried deep inside it can take a long time. Alternatively, Java allows a program to perform *random file access*. In random file access, a program can immediately jump to any location in the file without first reading the preceding bytes. The difference between sequential and random file access is like the difference between a cassette tape player and an MP3 player. When listening to an MP3 player, there is no need to listen to or fast-forward over unwanted songs. You simply jump to the track you want to hear. This is illustrated in [Figure 10-9](#).

## **Figure 10-9 Sequential access versus random access**

Items in a sequential access file are accessed one after the other.



Items in a random access file are accessed in any order.



[Figure 10-9 Full Alternative Text](#)

To create and work with random access files in Java, you use the `RandomAccessFile` class, which is in the `java.io` package. The general format of the class constructor is:

```
RandomAccessFile(String filename, String mode)
```

The first argument is the name of the file. The second argument is a string indicating the mode in which you wish to use the file. The two modes are "r" for reading, and "rw" for reading and writing. When a file is opened with "r" as the mode, the program can only read from the file. When a file is opened with "rw" as the mode, the program can read from the file and write to it. Here are some examples of statements that open files using the `RandomAccessFile` class:

```
// Open a file for random reading.
RandomAccessFile randomFile = new RandomAccessFile("MyData.dat",

// Open a file for random reading and writing.
RandomAccessFile randomFile = new RandomAccessFile("MyData.dat",
```

Here are some important points to remember about the two modes:

- If you open a file in "r" mode and the file does not exist, a

`FileNotFoundException` will be thrown.

- If you open a file in "r" mode and try to write to it, an `IOException` will be thrown.
- If you open an existing file in "rw" mode, it will not be deleted. The file's existing contents will be preserved.
- If you open a file in "rw" mode and the file does not exist, it will be created.

## Reading and Writing with the RandomAccessFile Class

A file that is opened or created with the `RandomAccessFile` class is treated as a binary file. In fact, the `RandomAccessFile` class has the same methods as the `DataOutputStream` class for writing data, and the same methods as the `DataInputStream` class for reading data. In fact, you can use the `RandomAccessFile` class to sequentially process a binary file. For example, the program in [Code Listing 10-15](#) opens a file named *Latters.dat*, then writes all of the letters of the alphabet to the file.

### Code Listing 10-15 (`WriteLetters.java`)

```
1 import java.io.*;
2
3 /**
4 * This program uses a RandomAccessFile object to create
5 * the file Letters.dat. The letters of the alphabet are
6 * written to the file.
7 */
8
9 public class WriteLetters
10 {
```

```

11 public static void main(String[] args) throws IOException
12 {
13 // The letters array has all 26 letters of the alphabet.
14 char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g',
15 'h', 'i', 'j', 'k', 'l', 'm', 'n',
16 'o', 'p', 'q', 'r', 's', 't', 'u',
17 'v', 'w', 'x', 'y', 'z' };
18
19 System.out.println("Opening the file.");
20
21 // Open a file for reading and writing.
22 RandomAccessFile randomFile =
23 new RandomAccessFile("Letters.dat", "rw");
24
25 System.out.println("Writing data to the file...");
26
27 // Sequentially write the letters array to the file.
28 for (int i = 0; i < letters.length; i++)
29 randomFile.writeChar(letters[i]);
30
31 // Close the file.
32 randomFile.close();
33 System.out.println("Done.");
34 }
35 }
```

## Program Output

```

Opening the file.
Writing data to the file...
Done.
```

After this program executes, the letters of the alphabet will be stored in the *Lettters.dat* file. Because the `writeChar` method was used, the letters will each be stored as two-byte characters. This fact will be important to know later when we want to read the characters from the file.

# The File Pointer

The `RandomAccessFile` class treats a file as a stream of bytes. The bytes are numbered, with the first byte being byte 0. The last byte's number is one less

than the number of bytes in the file. These byte numbers are similar to an array's subscripts, and are used to identify locations in the file.

Internally, the `RandomAccessFile` class keeps a long integer value known as the file pointer. The *file pointer* holds the byte number of a location in the file. When a file is first opened, the file pointer is set to 0. This causes it to “point” to the first byte in the file. When an item is read from the file, it is read from the byte to which the file pointer points. Reading also causes the file pointer to advance to the byte just beyond the item that was read. For example, let's say the file pointer points to byte 0, and an `int` is read from the file with the `readInt` method. An `int` is four bytes in size, so four bytes will be read from the file, starting at byte 0. After the value is read, the file pointer will be advanced to byte number 4, which is the fifth byte in the file. If another item is immediately read, the reading will begin at byte number 4. If the file pointer refers to a byte number that is beyond the end of the file, an `EOFException` is thrown when a read operation is performed.

Writing also takes place at the location pointed to by the file pointer. If the file pointer points to the end of the file when a write operation is performed, then the data will be written to the end of the file. However, if the file pointer holds the number of a byte within the file, at a location where data is already stored, then a write operation will cause data to be written over the existing data at that location.

Not only does the `RandomAccessFile` class let you read and write data, but it also allows you to move the file pointer. This means that you can immediately read data from any byte location in the file. It also means that you can write data to any location in the file, over existing data. To move the file pointer, you use the `seek` method. Here is the method's general format:

```
void seek(long position)
```

The argument is the number of the byte to which you want to move the file pointer. For example, look at the following code:

```
RandomAccessFile file = new RandomAccessFile("MyInfo.dat", "r");
file.seek(99);
byte b = file.readByte();
```

This code opens the file *MyInfo.dat* for reading. The `seek` method is called to move the file pointer to byte number 99 (which is the 100th byte in the file). Then, the `readByte` method is called to read byte number 99 from the file. After that statement executes, the file pointer will be advanced by one byte, so it will point to byte 100. Suppose we continue processing the same file with the following code:

```
file.seek(49);
int i = file.readInt();
```

First, the `seek` method moves the file pointer to byte number 49 (which is the 50th byte in the file). Then, the `readInt` method is called. This reads an `int` from the file. An `int` is four bytes in size, so this statement reads four bytes, beginning at byte number 49. After the statement executes the file pointer will be advanced by four bytes, so it will point to byte 53.

Although a file might contain `chars`, `ints`, `doubles`, `strings`, and so forth, the `RandomAccessFile` class sees it only as a stream of bytes. The class is unaware of the data types of the data stored in the file, and it cannot determine where one item of data ends and another begins. When you write a program that reads data from a random access file, it is your responsibility to know how the data is structured.

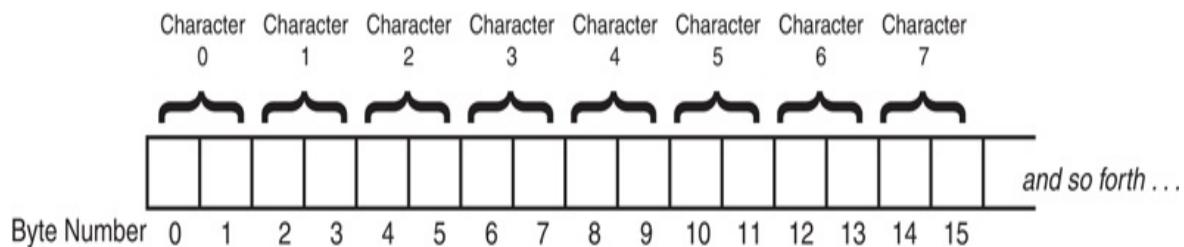
For example, recall that the program in [Code Listing 10-14](#) wrote the letters of the alphabet to the *Latters.dat* file. Let's say the first letter is character 0, the second letter is character 1, and so forth. Suppose we want to read character 5 (the sixth letter in the file). At first, we might be tempted to try the following code:

```
// Open the file for reading.
RandomAccessFile randomFile = new RandomAccessFile("Letters.dat",
// Move the file pointer to byte 5, which is the 6th byte.
randomFile.seek(5);
// Read the character.
char ch = randomFile.readChar();
// What will this display?
System.out.println("The sixth letter is " + ch);
```

Although this code will compile and run, you might be surprised at the result. Recall that the `writeChar` method writes a character as two bytes. Because

each character occupies two bytes in the file, the sixth character begins at byte 10, not byte 5. This is illustrated in [Figure 10-10](#). In fact, if we try to read a character starting at byte 5, we will read garbage because byte 5 is not at the beginning of a character.

## Figure 10-10 Layout of the *Letters.dat* file



[Figure 10-10 Full Alternative Text](#)

To determine the position of a character in the file, we must take each character's size into account. The following code will correctly read and display the sixth character. To determine the character's starting byte number, it multiplies the size of a character by the number of the character we want to locate.

```
final int CHAR_SIZE = 2; // Each char uses two bytes
// Move the file pointer to character 5.
randomFile.seek(CHAR_SIZE * 5);
// Read the character.
char ch = randomFile.readChar();
// This will display the correct character.
System.out.println("The sixth character is " + ch);
```

The program in [Code Listing 10-16](#) demonstrates further. It randomly reads characters 5, 10, and 3 from the file.

## Code Listing 10-16

# (ReadRandomLetters.java)

```
1 import java.io.*;
2
3 /**
4 * This program uses the RandomAccessFile class
5 * to open the file Letters.dat and randomly read
6 * letters from different locations.
7 */
8
9 public class ReadRandomLetters
10 {
11 public static void main(String[] args) throws IOException
12 {
13 final int CHAR_SIZE = 2; // 2 byte characters
14 long byteNum; // For the byte number
15 char ch; // To hold a character
16
17 // Open the file for reading.
18 RandomAccessFile randomFile =
19 new RandomAccessFile("Letters.dat", "r");
20
21 // Move to character 5. This is the sixth character
22 // from the beginning of the file.
23 byteNum = CHAR_SIZE * 5;
24 randomFile.seek(byteNum);
25
26 // Read the character stored at this location
27 // and display it. Should be the letter f.
28 ch = randomFile.readChar();
29 System.out.println(ch);
30
31 // Move to character 10 (the 11th character),
32 // read the character and display it.
33 // Should be the letter k.
34 byteNum = CHAR_SIZE * 10;
35 randomFile.seek(byteNum);
36 ch = randomFile.readChar();
37 System.out.println(ch);
38
39 // Move to character 3 (the fourth character),
40 // read the character and display it.
41 // Should be the letter d.
42 byteNum = CHAR_SIZE * 3;
43 randomFile.seek(byteNum);
44 ch = randomFile.readChar();
```

```
45 System.out.println(ch);
46
47 // Close the file.
48 randomFile.close();
49 }
50 }
```

## Program Output

f  
k  
d

Appendix H *Working with Records and Random Access Files* is available on this book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

# Object Serialization

In the previous section, you saw how an object's fields can be retrieved and saved to a file as fields in a record. If an object contains other types of objects as fields, however, the process of saving its contents can become complicated. Fortunately, Java allows you to *serialize* objects, which is a simpler way of saving objects to a file.

When an object is serialized, it is converted into a series of bytes that contain the object's data. If the object is set up properly, even the other objects that it might contain as fields are automatically serialized. The resulting set of bytes can be saved to a file for later retrieval.

In order for an object to be serialized, its class must implement the `Serializable` interface. The `Serializable` interface, which is in the `java.io` package, has no methods or fields. It is used only to let the Java compiler know that objects of the class might be serialized. In addition, if a class contains objects of other classes as fields, those classes must also implement the `Serializable` interface, in order to be serialized.

For example, the book's source code contains a modified version of the `InventoryItem` class named `InventoryItem2`. The only modification to the

class is that it implements the `Serializable` interface. Here are the modified lines of code from the file:

```
import java.io.Serializable;
public class InventoryItem2 implements Serializable
```

This new code tells the compiler that we want to be able to serialize objects of the `InventoryItem2` class. But what about the class's `description` field, which is `String` object? The `String` class, as well as many others in the Java API, also implement the `Serializable` interface. So, the `InventoryItem2` class is ready for serialization.

To write a serialized object to a file, you use an `ObjectOutputStream` object. The `ObjectOutputStream` class is designed to perform the serialization process (converting an object to a series of bytes). To write the bytes to a file, you must also use an output stream object, such as `FileOutputStream`. Here is an example:

```
FileOutputStream outStream = new FileOutputStream("Objects.dat");
ObjectOutputStream objectOutputFile = new ObjectOutputStream(outS
```

To serialize an object and write it to the file, use the `ObjectOutputStream` class's `writeObject` method, as shown here:

```
InventoryItem2 item = new InventoryItem2("wrench", 20);
objectOutputFile.writeObject(item);
```

The `writeObject` method throws an `IOException` if an error occurs.

The process of reading a serialized object's bytes and constructing an object from them is known as *deserialization*. To deserialize an object, you use an `ObjectInputStream` object, along with a `FileInputStream` object. Here is an example of how to set the objects up:

```
FileInputStream inStream = new FileInputStream("Objects.dat");
ObjectInputStream objectInputFile = new ObjectInputStream(inStrea
```

To read a serialized object from the file, use the `ObjectInputStream` class's `readObject` method. Here is an example:

```
InventoryItem2 item;
item = (InventoryItem2) objectInputStream.readObject();
```

The `readObject` method returns the deserialized object. Notice you must cast the return value to the desired class type. (The `readObject` method throws a number of different exceptions if an error occurs. See the API documentation for more information.)

The following programs demonstrate how to serialize and deserialize objects. The program in [Code Listing 10-17](#) serializes three `InventoryItem2` objects, and the program in [Code Listing 10-18](#) deserializes them.

## Code Listing 10-17 (`SerializeObjects.java`)

```
1 import java.util.Scanner;
2 import java.io.*;
3
4 /**
5 * This program serializes the objects in an array of
6 * InventoryItem2 objects.
7 */
8
9 public class SerializeObjects
10 {
11 public static void main(String[] args) throws IOException
12 {
13 final int NUM_ITEMS = 3; // Number of items
14 String description; // Item description
15 int units; // Units on hand
16
17 // Create a Scanner object for keyboard input.
18 Scanner keyboard = new Scanner(System.in);
19
20 // Create an array to hold InventoryItem objects.
21 InventoryItem2[] items =
22 new InventoryItem2[NUM_ITEMS];
23
24 // Get data for the InventoryItem objects.
25 System.out.println("Enter data for " + NUM_ITEMS +
26 " inventory items.");
27 }
}
```

```

28 for (int i = 0; i < items.length; i++)
29 {
30 // Get the item description.
31 System.out.print("Enter an item description: ");
32 description = keyboard.nextLine();
33
34 // Get the number of units.
35 System.out.print("Enter the number of units: ");
36 units = keyboard.nextInt();
37
38 // Consume the remaining newline.
39 keyboard.nextLine();
40
41 // Create an InventoryItem2 object in the array.
42 items[i] = new InventoryItem2(description, units);
43 }
44
45 // Create the stream objects.
46 FileOutputStream outStream =
47 new FileOutputStream("Objects.dat");
48 ObjectOutputStream objectOutputStream =
49 new ObjectOutputStream(outStream);
50
51 // Write the serialized objects to the file.
52 for (int i = 0; i < items.length; i++)
53 {
54 objectOutputStream.writeObject(items[i]);
55 }
56
57 // Close the file.
58 objectOutputStream.close();
59 System.out.println("The serialized objects were written to
60 "Objects.dat file.");
61 }
62 }
```

## Program Output with Example Input Shown in Bold

Enter data for 3 inventory items.

Enter an item description: **Wrench** 

Enter the number of units: **20** 

Enter an item description: **Hammer** 

Enter the number of units: **15** 

Enter an item description: **Pliers** 

Enter the number of units: 12   
The serialized objects were written to the Objects.dat file.

## Code Listing10-18 (**DeserializeObjects.java**)

```
1 import java.io.*;
2
3 /**
4 * This program deserializes the objects in the Objects.dat
5 * file and stores them in an array.
6 */
7
8 public class DeserializeObjects
9 {
10 public static void main(String[] args) throws Exception
11 {
12 final int NUM_ITEMS = 3; // Number of items
13
14 // Create the stream objects.
15 FileInputStream inStream =
16 new FileInputStream("Objects.dat");
17 ObjectInputStream objectInputFile =
18 new ObjectInputStream(inStream);
19
20 // Create an array to hold InventoryItem objects.
21 InventoryItem2[] items = new InventoryItem2[NUM_ITEMS];
22
23 // Read the serialized objects from the file.
24 for (int i = 0; i < items.length; i++)
25 {
26 items[i] =
27 (InventoryItem2) objectInputFile.readObject();
28 }
29
30 // Close the file.
31 objectInputFile.close();
32
33 // Display the objects.
34 for (int i = 0; i < items.length; i++)
35 {
36 System.out.println("Item " + (i + 1));
37 System.out.println(" Description: " +
```

```
38 items[i].getDescription());
39 System.out.println(" Units: " +
40 items[i].getUnits());
41 }
42 }
43 }
```

## Program Output

```
Item 1
Description: Wrench
Units: 20
Item 2
Description: Hammer
Units: 15
Item 3
Description: Pliers
Units: 12
```



## Checkpoint

1. 10.20 What is the difference between a text file and a binary file?
2. 10.21 What classes do you use to write output to a binary file? To read from a binary file?
3. 10.22 What is the difference between sequential and random access?
4. 10.23 What class do you use to work with random access files?
5. 10.24 What are the two modes a random access file can be opened in? Explain the difference between them.
6. 10.25 What must you do to a class in order to serialize objects of that class?

## 10.4 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics.

- Assuming that all statements inside a try block will execute. When an exception is thrown, the try block is exited immediately. This means that statements appearing in the try block after the offending statement will not be executed.
- Getting the try, catch, and finally clauses out of order. In a `try` statement, the `try` clause must appear first, followed by all of the `catch` clauses, followed by the optional `finally` clause.
- Writing two catch clauses that handle the same exception in the same try statement. You cannot have more than one catch clause per exception type in the same `try` statement.
- When catching multiple exceptions that are related to one another through inheritance, listing the more general exceptions first. When in the same `try` statement you are handling multiple exceptions, and some of the exceptions are related to each other through inheritance, you should handle the more specialized exception classes before the more general exception classes. Otherwise, an error will occur because the compiler thinks you are handling the same exception more than once.
- Forgetting to write a `throws` clause on a method that can throw a checked exception but does not handle the exception. If a method is capable of throwing a checked exception but does not handle the exception, it must have a `throws` clause in its header that specifies the exception.
- Calling a method but not handling an exception that it might throw. You must either handle all of the checked exceptions that a method can throw, or list them in the calling method's `throws` clause.

- In a custom exception class, forgetting to pass an error message to the superclass's constructor. If you do not pass an error message to the superclass's constructor, the exception object will have a null error message.
- Serializing an object with members that are not serializable. If a class has fields that are objects of other classes, those classes must implement the `Serializable` interface in order to be serialized.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. When an exception is generated, it is said to have been
  1. built
  2. thrown
  3. caught
  4. killed
2. This is a section of code that gracefully responds to exceptions.
  1. exception generator
  2. exception manipulator
  3. exception handler
  4. exception monitor
3. If your code does not handle an exception when it is thrown, it is dealt with by this.
  1. default exception handler
  2. the operating system
  3. system debugger
  4. default exception generator

4. All exception classes inherit from this class.
  1. Error
  2. RuntimeException
  3. JavaException
  4. Throwable
5. `FileNotFoundException` inherits from .
  1. Error
  2. IOException
  3. JavaException
  4. FileException
6. You can think of this code as being “protected” because the application will not halt if it throws an exception.
  1. try block
  2. catch block
  3. finally block
  4. protected block
7. This method can be used to retrieve the error message from an exception object.
  1. errorMessage
  2. errorString
  3. getError

- 4. `getMessage`
- 8. The numeric wrapper classes’ “parse” methods all throw an exception of this type.
  - 1. `ParseException`
  - 2. `NumberFormatException`
  - 3. `IOException`
  - 4. `BadNumberException`
- 9. This is one or more statements that are always executed after the try block has executed, and after any catch blocks have executed if an exception was thrown.
  - 1. try block
  - 2. catch block
  - 3. finally block
  - 4. protected block
- 10. This is an internal list of all the methods that are currently executing.
  - 1. invocation list
  - 2. call stack
  - 3. call list
  - 4. list trace
- 11. This method can be called from any exception object, and it shows the chain of methods that were called when the exception was thrown.
  - 1. `printInvocationList`

2. `printCallStack`
  3. `printStackTrace`
  4. `printCallList`
12. These are exceptions that inherit from the `Error` class or the `RuntimeException` class.
1. unrecoverable exceptions
  2. unchecked exceptions
  3. recoverable exceptions
  4. checked exceptions
13. All exceptions that do *not* inherit from the `Error` class or the `RuntimeException` class are .
1. unrecoverable exceptions
  2. unchecked exceptions
  3. recoverable exceptions
  4. checked exceptions
14. This informs the compiler of the exceptions that could get thrown from a method.
1. `throws` clause
  2. parameter list
  3. catch clause
  4. method return type

15. You use this statement to manually throw an exception.
1. try
  2. generate
  3. throw
  4. `System.exit(0)`
16. This is the process of converting an object to a series of bytes that represent the object's data.
1. Serialization
  2. Deserialization
  3. Dynamic conversion
  4. Casting
17. True or False: You are not required to catch exceptions that inherit from the `RuntimeException` class.
18. True or False: When an exception is thrown by code inside a try block, all of the statements in the try block are always executed.
19. True or False: `IOException` serves as a superclass for exceptions related to programming errors, such as an out-of-bounds array subscript.
20. True or False: You cannot have more than one catch clause per try statement.
21. True or False: When an exception is thrown, the JVM searches the try statement's catch clauses from top to bottom, and passes control of the program to the first catch clause with a parameter that is compatible with the exception.
22. True or False: Not including polymorphic references, a try statement

can have only one catch clause for each specific type of exception.

23. True or False: When in the same try statement you are handling multiple exceptions, and some of the exceptions are related to each other through inheritance, you should handle the more general exception classes before the more specialized exception classes.
24. True or False: The throws clause causes an exception to be thrown.

## Find the Error

Find the error in each of the following code segments:

1. `catch (FileNotFoundException e)`

```
{
 System.out.println("File not found.");
}
try
{
 File file = new File("MyFile.txt");
 Scanner inputFile = new Scanner(file);
}
```

2. `// Assume inputFile references a Scanner object.`

```
try
{
 input = inputFile.nextInt();
}
finally
{
 inputFile.close();
}
catch (InputMismatchException e)
{
 System.out.println(e.getMessage());
}
```

3. `try`

```

{
 number = Integer.parseInt(str);
}
catch (Exception e)
{
 System.out.println(e.getMessage());
}
catch (IllegalArgumentException e)
{
 System.out.println("Bad number format.");
}
catch (NumberFormatException e)
{
 System.out.println(str + " is not a number.");
}

```

# Algorithm Workbench

1. Look at the following program and tell what the program will output when run:

```

public class ExceptionTest
{
 public static void main(String[] args)
 {
 int number;
 String str;

 try
 {
 str = "xyz";
 number = Integer.parseInt(str);
 System.out.println("A");
 }
 catch(NumberFormatException e)
 {
 System.out.println("B");
 }
 catch(IllegalArgumentException e)
 {
 System.out.println("C");
 }

 System.out.println("D");
 }
}

```

```
 }
}
```

2. Look at the following program and tell what the program will output when run:

```
public class ExceptionTest
{
 public static void main(String[] args)
 {
 int number;
 String str;

 try
 {
 str = "xyz";
 number = Integer.parseInt(str);
 System.out.println("A");
 }
 catch(NumberFormatException e)
 {
 System.out.println("B");
 }
 catch(IllegalArgumentException e)
 {
 System.out.println("C");
 }
 finally
 {
 System.out.println("D");
 }
 System.out.println("E");
 }
}
```

3. Write a method that searches a numeric array for a specified value. The method should return the subscript of the element containing the value if it is found in the array. If the value is not found, the method should throw an exception of the `Exception` class with the error message “Element not found”.
4. Write a statement that throws an `IllegalArgumentException` with the error message “Argument cannot be negative”.

5. Write an exception class that can be thrown when a negative number is passed to a method.
6. Write a statement that throws an instance of the exception class you created in Question 5.
7. The method `getValueFromFile` is public and returns an `int`. It accepts no arguments. The method is capable of throwing an `IOException` and a `FileNotFoundException`. Write the header for this method.
8. Write a `try` statement that calls the `getValueFromFile` method described in Question 7. Be sure to handle all the exceptions that the method can throw.
9. Write a statement that creates an object that can be used to write binary data to the file *Configuration.dat*.
10. Assume the reference variable `r` refers to a serializable object. Write code that serializes the object to the file *ObjectData.dat*.

## Short Answer

1. What is meant when it is said that an exception is thrown?
2. What does it mean to catch an exception?
3. What happens when an exception is thrown, but the `try` statement does not have a `catch` clause that is capable of catching it?
4. What is the purpose of a `finally` clause?
5. Where does execution resume after an exception has been thrown and caught?
6. When multiple exceptions are caught in the same `try` statement, and some of them are related through inheritance, does the order in which they are listed matter?

7. What types of objects can be thrown?
8. When are you required to have a throws clause in a method header?
9. What is the difference between a checked exception and an unchecked exception?
10. What is the difference between the throw statement and the throws clause?
11. What is the difference between a text file and a binary file?
12. What is the difference between a sequential access file and a random access file?
13. What happens when you serialize an object? What happens when you deserialize an object?

# Programming Challenges

## 1. TestScores Class

Write a class named `TestScores`. The class constructor should accept an array of test scores as its argument. The class should have a method that returns the average of the test scores. If any test score in the array is negative or greater than 100, the class should throw an `IllegalArgumentException`. Demonstrate the class in a program.

## 2. TestScores Class Custom Exception

Write an exception class named `InvalidTestScore`. Modify the `TestScores` class you wrote in Programming Challenge 1 so it throws an `InvalidTestScore` exception if any of the test scores in the array are invalid.

## 3. RetailItem Exceptions

Programming Challenge 4 of [Chapter 3](#) required you write a `RetailItem` class that held data pertaining to a retail item. Write an exception class that can be instantiated and thrown when a negative number is given for the price. Write another exception class that can be instantiated and thrown when a negative number is given for the units on hand. Demonstrate the exception classes in a program.

## 4. Month Class Exceptions

Programming Challenge 5 of [Chapter 6](#) required you write a `Month` class that holds information about the month. Write exception classes for the following error conditions:

- A number less than 1 or greater than 12 is given for the month number.
- An invalid string is given for the name of the month.

Modify the Month class so it throws the appropriate exception when either of these errors occurs. Demonstrate the classes in a program.

## 5. Payroll Class Exceptions

Programming Challenge 5 of [Chapter 3](#) required you write a Payroll class that calculates an employee's payroll. Write exception classes for the following error conditions:

- An empty string is given for the employee's name.
- An invalid value is given for the employee's ID number. If you implemented this field as a string, then an empty string would be invalid. If you implemented this field as a numeric variable, then a negative number or zero would be invalid.
- An invalid number is given for the number of hours worked. This would be a negative number, or a number greater than 84.
- An invalid number is given for the hourly pay rate. This would be a negative number, or a number greater than 25.

Modify the Payroll class so it throws the appropriate exception when any of these errors occurs. Demonstrate the exception classes in a program.

## 6. FileArray Class

Design a class that has a static method named `writeArray`. The method should take two arguments: the name of a file and a reference to an `int` array. The file should be opened as a binary file, the contents of the array should be written to the file, then the file should be closed.

Write a second method in the class named `readArray`. The method should take two arguments: the name of a file, and a reference to an `int` array. The file should be opened, data should be read from the file and stored in the array, then the file should be closed. Demonstrate both methods in a program.

## 7. File Encryption Filter

File encryption is the science of writing the contents of a file in a secret code. Your encryption program should work like a filter, reading the contents of one file, modifying the data into a code, then writing the coded contents out to a second file. The second file will be a version of the first file, but written in a secret code.

Although there are complex encryption techniques, you should come up with a simple one of your own. For example, you could read the first file one character at a time, and add 10 to the character code of each character before it is written to the second file.

## 8. File Decryption Filter

Write a program that decrypts the file produced by the program in Programming Challenge 7. The decryption program should read the contents of the coded file, restore the data to its original state, and write it to another file.

## 9. TestScores Modification for Serialization

Modify the `TestScores` class that you created for Programming Challenge 1 to be serializable. Write a program that creates an array of at least five `TestScore` objects and serializes them. Write another program that deserializes the objects from the file.

## 10. Bank Account Random File



### Note:

To do this assignment, be sure to read Appendix H, available on this book's online resource page at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

One of the example classes you saw in this chapter was the `BankAccount` class. Write a `BankAccountFile` class that manages a random access file

of BankAccount object records. The class should read and write records, and move the file pointer to any location within the file. (The class should be similar to the `InventoryItemFile` class presented in Appendix H.) In a program or programs, demonstrate how to create records, randomly look at records, and randomly modify records.

## 11. Exception Project

This assignment assumes you have completed Programming Challenge 1 of [Chapter 9](#) (Employee and ProductionWorker Classes). Modify the Employee and ProductionWorker classes so they throw exceptions when the following errors occur:

- The Employee class should throw an exception named `InvalidEmployeeNumber` when it receives an employee number that is less than 0 or greater than 9999.
- The ProductionWorker class should throw an exception named `InvalidShift` when it receives an invalid shift.
- The ProductionWorker class should throw an exception named `InvalidPayRate` when it receives a negative number for the hourly pay rate.

Write a test program that demonstrates how each of these exception conditions work.



**VideoNote** The Exception Project Problem

# **Chapter 11 JavaFX: GUI Programming and Basic Controls**

## **Topics**

1. [11.1 Graphical User Interfaces](#)
2. [11.2 Introduction to JavaFX](#)
3. [11.3 Creating Scenes](#)
4. [11.4 Displaying Images](#)
5. [11.5 More about the HBox, VBox, and GridPane Layout Containers](#)
6. [11.6 Button Controls and Events](#)
7. [11.7 Reading Input with TextField Controls](#)
8. [11.8 Using Lambda Expressions to Handle Events](#)
9. [11.9 The BorderPane Layout Container](#)
10. [11.10 The ObservableList Interface](#)
11. [11.11 Common Errors to Avoid](#)

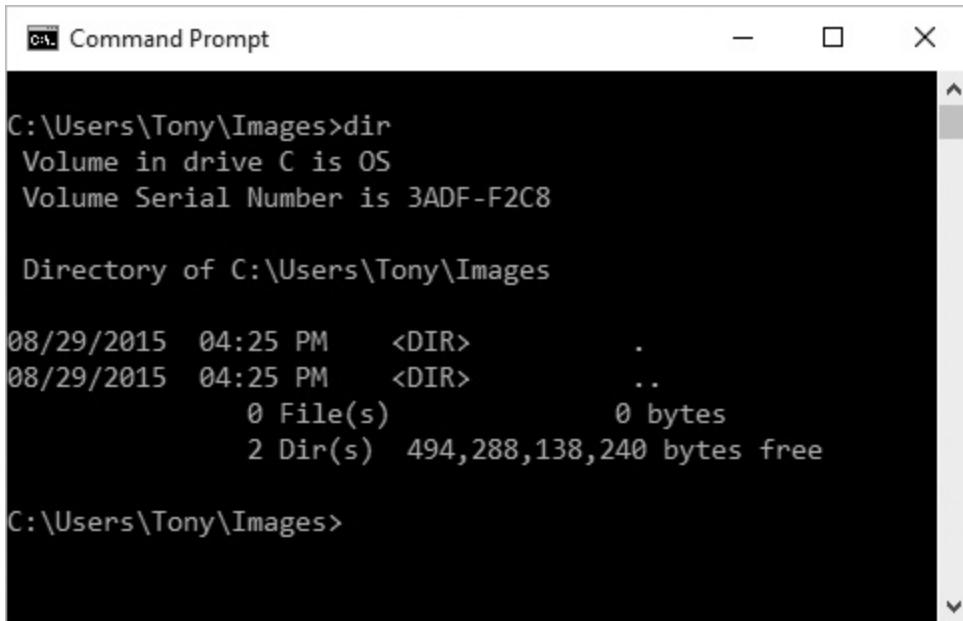
# **11.1 Graphical User Interfaces**

## **Concept:**

A graphical user interface allows the user to interact with a program using graphical elements such as windows, icons, buttons, and so on.

Programmers commonly employ the term *user* to describe any hypothetical person who might be using a computer and its programs. A computer's *user interface* is the part of the computer with which the user interacts. One part of the user interface consists of hardware devices, such as the keyboard and video display. Another part of the user interface involves the way the computer's operating system and application software accepts commands from the user. For many years, the only way that the user could interact with a computer was through a command line interface. A *command line interface*, which is also known as a *console interface*, requires the user to type commands. If a command is typed correctly, it is executed and the results are displayed. If a command is not typed correctly, an error message is displayed. [Figure 11-1](#) shows the Windows command prompt window, which is an example of a command line interface.

## **Figure 11-1 A command line interface**



```
C:\Users\Tony\Images>dir
Volume in drive C is OS
Volume Serial Number is 3ADF-F2C8

Directory of C:\Users\Tony\Images

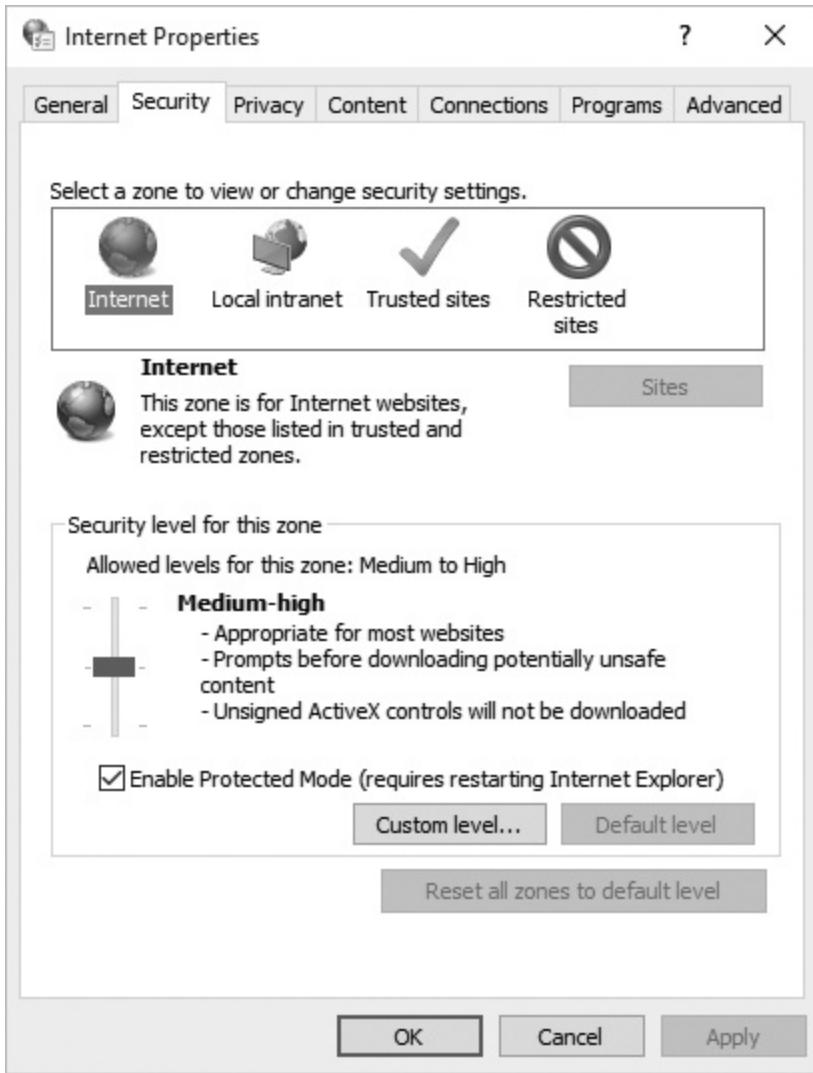
08/29/2015 04:25 PM <DIR> .
08/29/2015 04:25 PM <DIR> ..
 0 File(s) 0 bytes
 2 Dir(s) 494,288,138,240 bytes free

C:\Users\Tony\Images>
```

Many computer users, especially beginners, find command line interfaces difficult to use. This is because there are many commands to be learned, and each command has its own syntax, much like a programming statement. If a command isn't entered correctly, it will not work.

In the 1980s, a new type of interface known as a graphical user interface came into use in commercial operating systems. A *graphical user interface*, or *GUI* (pronounced “gooey”), allows the user to interact with the operating system and application programs through graphical elements on the screen. GUIs also popularized the use of the mouse as an input device. Instead of requiring the user to type commands on the keyboard, GUIs allow the user to point at graphical elements and click the mouse button to activate them. Much of the interaction with a GUI is done through windows that display information and allow the user to perform actions. [Figure 11-2](#) shows an example of a window that allows the user to change the system’s Internet settings. Instead of typing cryptic commands, the user interacts with icons, buttons, and slider bars.

## Figure 11-2 A window in a graphical user interface



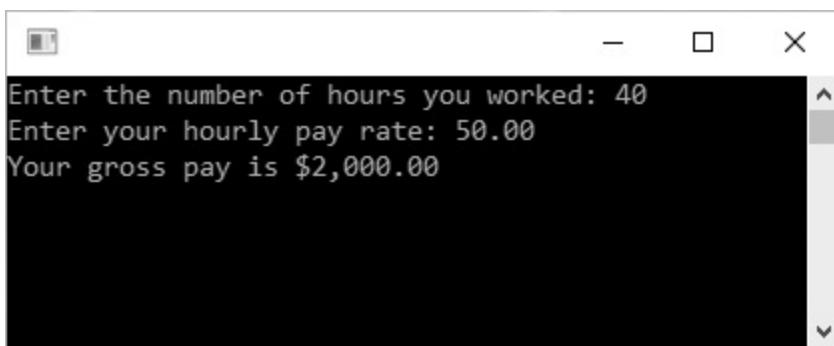
[Figure 11-2 Full Alternative Text](#)

## Event-Driven GUI Programs

In a text-based environment, such as a command line interface, programs determine the order in which things happen. For example, [Figure 11-3](#) shows the interaction that has taken place in a text environment with a program that calculates an employee's gross pay. First, the program prompted the user to enter the number of hours worked. In the figure, the user entered 40 and pressed the Enter key. Next, the program prompted the user to enter his or her hourly pay rate. In the figure, the user entered 50.00, and pressed the Enter key. Then, the program displayed the user's gross pay. As the program was

running, the user had no choice but to enter the data in the order requested.

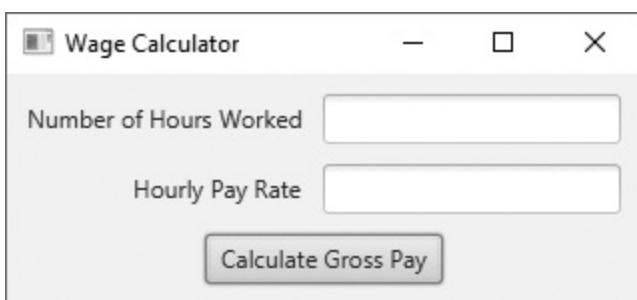
## Figure 11-3 Interaction with a program in a text environment



[Figure 11-3 Full Alternative Text](#)

In a GUI environment, however, the user determines the order in which things happen. For example, [Figure 11-4](#) shows a GUI application that calculates an employee's gross pay. Notice there are boxes into which the user enters the number of hours worked and the hourly pay rate. The user can enter the hours and the pay rate in any order he or she wishes. If the user makes a mistake, he or she can erase the data that was entered and retype it. When the user is ready to calculate the area, he or she clicks the *Calculate Gross Pay* button, and the program performs the calculation.

## Figure 11-4 A GUI application



Because GUI programs must respond to the actions of the user, they are said to be *event-driven*. The user causes events, such as the clicking of a button, and the program responds to those events.

This chapter focuses exclusively on the development of GUI applications using the JavaFX library. As you work through this chapter, you will learn to create applications that interact with the user through windows containing graphical objects. You will also learn how to program your applications to respond to the events that take place as the user interacts with them.



## Checkpoint

1. 11.1 What is a user interface?
2. 11.2 How does a command line interface work?
3. 11.3 When the user runs a program in a text-based environment, such as the command line interface, what determines the order in which things happen?
4. 11.4 What is an event-driven program?

## 11.2 Introduction to JavaFX

### Concept:

In Java, you can use the JavaFX library to create GUI and graphical applications. JavaFX is the next generation GUI toolkit for Java developers.

In this chapter, we will use JavaFX to create GUI applications. *JavaFX* is a standard Java library for developing rich applications that employ graphics. You can use it to create GUI applications, as well as applications that display 2D and 3D graphics. You can use JavaFX to create standalone graphics applications that run on your local computer, applications that run from a remote server, or applications that are embedded in a Web page. This chapter introduces you to JavaFX as a tool for creating standalone GUI applications.

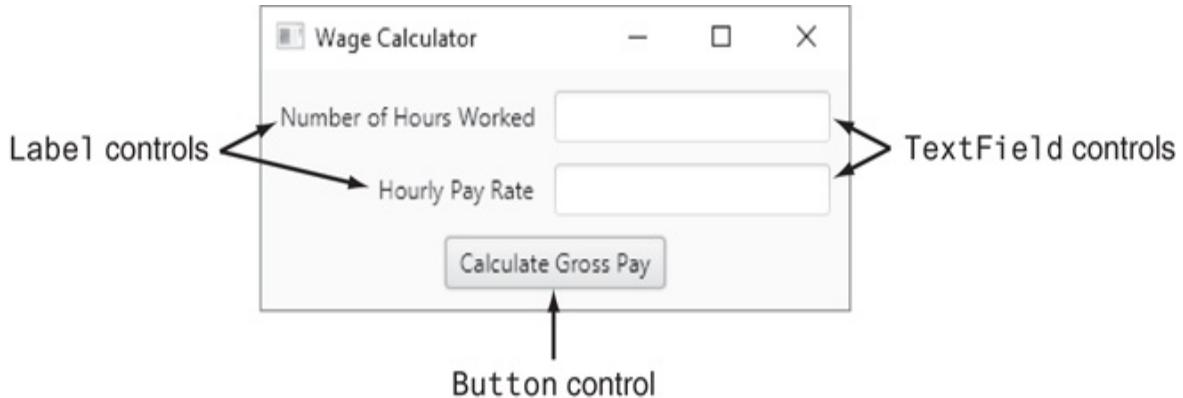


**VideoNote** Introduction to JavaFX

### Controls

The objects that are visible in a program's graphical user interface are commonly referred to as *controls*. The GUI shown in [Figure 11-5](#) contains two Label controls, two TextField controls, and a Button control. Here is a summary of each of these types of controls:

### Figure 11-5 Controls in a GUI



[Figure 11-5 Full Alternative Text](#)

- Label control A Label control displays text. The window shown in [Figure 11-5](#) contains two Label controls. One of the Label controls displays the text *Number of Hours Worked*, and the other Label control displays the text *Hourly Pay Rate*.
- TextField control A TextField control appears as a rectangular region that can accept keyboard input from the user. The window shown in [Figure 11-5](#) has two TextField controls: one in which the user enters the number of hours worked, and another in which the user enters the hourly pay rate.
- Button control A Button is a rectangular control that appears as a button with a caption written across its face. When the user clicks a Button control (either with a mouse or by touching it on a touch screen), an action takes place. The window in [Figure 11-5](#) has one Button control, showing the caption *Calculate Gross Pay*. When the user clicks this button, the program calculates and displays the gross pay.

Labels, TextFields, and Buttons are just a few of the controls you will learn to use. As you study this chapter, you will create applications that incorporate many different types of controls.

## Stages and Scenes

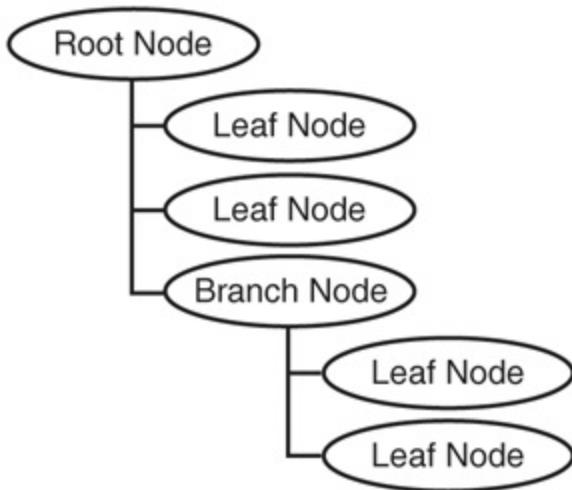
When programming with JavaFX, you use a theater metaphor to describe the elements of a GUI. Imagine that you are seated in a theater watching a play.

There is a stage, and on that stage, a scene is being performed.

A JavaFX application also has a stage and a scene. The stage is a window, and the scene is a collection of GUI objects (Labels, TextFields, Buttons, and other controls) contained within the window. You can think of the GUI objects as the actors that make up the scene.

In memory, the GUI objects in a scene are organized as nodes in a *scene graph*, which is a tree-like hierarchical data structure. [Figure 11-6](#) shows an example.

## Figure 11-6 Nodes in a scene graph



[Figure 11-6 Full Alternative Text](#)

A scene graph can have three types of nodes:

- Root Node: There is only one root node, which is the parent of all the other nodes in the scene graph.
- Branch Node: A branch node can have other nodes as children.

- Leaf Node: A leaf node cannot have children.

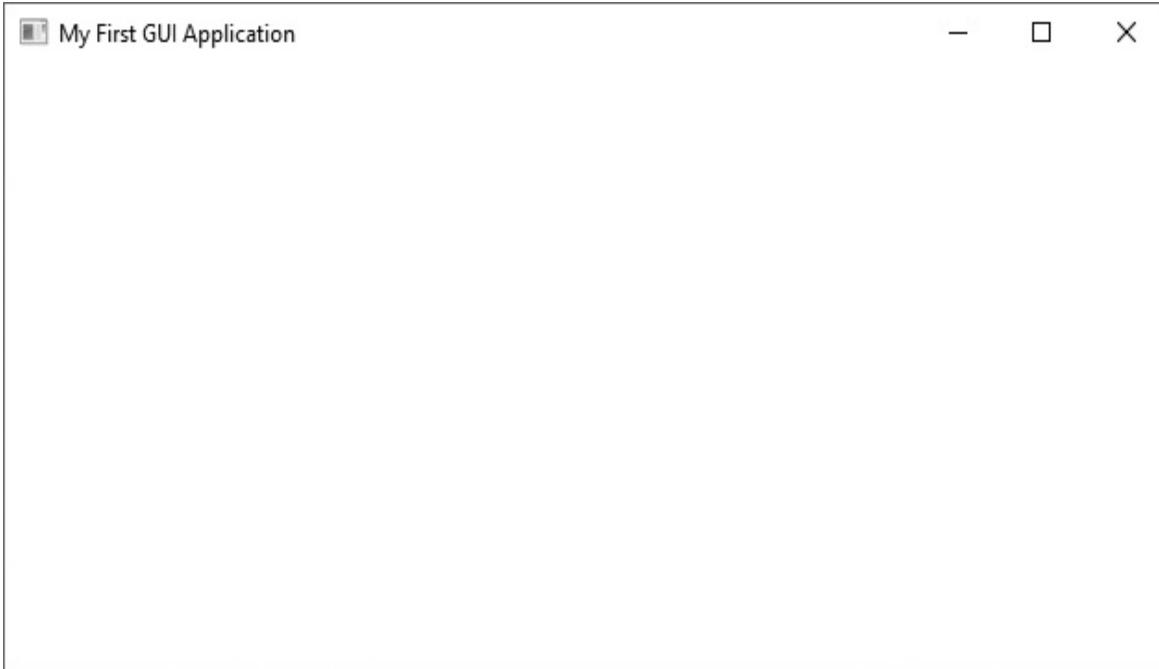
In a nutshell, the root node and branch nodes can have children, but leaf nodes cannot. Leaf nodes are typically controls with which the user interacts, such as Labels, TextFields, and Buttons. Branch nodes are objects that serve as containers for other nodes.

## The Application Class

The JavaFX library has an abstract class named `Application` that is the foundation of a GUI application. All JavaFX applications must extend the `Application` class.

The `Application` class has an abstract method named `start`, which is the entry point for the application. Because the `start` method is abstract, you must override it. Let's look at an example. The program in [Code Listing 11-1](#) is a simple JavaFX application that displays an empty window, as shown in [Figure 11-7](#). The program creates a stage, but does not create a scene.

**Figure 11-7 Window displayed  
by MyFirstGUI.java**



## Code Listing 11-1 (**MyFirstGUI.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3
4 /**
5 * A simple JavaFX GUI application
6 */
7
8 public class MyFirstGUI extends Application
9 {
10 public static void main(String[] args)
11 {
12 // Launch the application.
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Set the stage title.
20 primaryStage.setTitle("My First GUI Application");
21 }
}
```

```
22 // Show the window.
23 primaryStage.show();
24 }
25 }
```

Let's take a closer look at the program in [Code Listing 11-1](#):

- Line 1 imports the `Application` class, which is in the `javafx.application` package.
- Line 2 imports the `Stage` class, which is in the `javafx.stage` package. (You will see this class used in the `start` method.)
- The declaration for the `MyFirstGUI` class begins in line 8. Notice that it extends the `Application` class.
- The `main` method appears in lines 10 through 14. The `main` method does only one thing: it calls the `launch` method in line 13. The `launch` method, which is inherited from the `Application` class, does a number of setup operations, including the following:
  - It creates a `Stage` object that will be the application's window.
  - It calls the `start` method, passing a reference to the `Stage` object as an argument.
- The `start` method appears in lines 17 through 24. This is the entry point for the application. As previously mentioned, the `start` method is an abstract method in the `Application` class, and we must override it. Notice the method has a parameter named `primaryStage`. The `primaryStage` variable will reference the `Stage` object created by the `launch` method.
- Line 20 calls the `Stage` object's `setTitle` method. This sets the text to be displayed in the window's title bar.
- Line 23 calls the `Stage` object's `show` method. This displays the application's window.



# Checkpoint

1. 11.5 What is JavaFX?
2. 11.6 Is a window the same as a stage, or a scene?
3. 11.7 What is a scene graph?
4. 11.8 What are the three types of nodes in a scene graph?
5. 11.9 What type of scene graph node can have children? What type cannot?
6. 11.10 The JavaFX library has an abstract class that is the foundation of a GUI application. What is the name of the class?
7. 11.11 What is the purpose of the `launch` method of the `Application` class?
8. 11.12 What is the purpose of the `Application` class's abstract `start` method?
9. 11.13 The program in [Code Listing 11-1](#) calls a `Stage` class's `setTitle` method. What does this method do?
10. 11.14 The program in [Code Listing 11-1](#) calls a `Stage` class's `show` method. What does this method do?

# 11.3 Creating Scenes

## Concept:

A scene is a collection of controls and other objects in a JavaFX GUI.

Recall that we use a theater metaphor when describing a JavaFX GUI. In JavaFX terminology, a window is called a stage, and the collection of GUI objects that is displayed in that window is called a scene. The program you saw in [Code Listing 11-1](#) created a stage, but it did not create a scene. As a result, the program displays an empty window. Now, it's time to learn how to create a scene that contains JavaFX controls.

In general, the steps that you follow when creating a scene are:

1. Create the controls that will be in the scene.
2. Create a layout container of some type, and add the controls to the layout container.
3. Create a Scene object, and add the layout container to the Scene object.
4. Add the Scene object to the stage.

Let's take a closer look at each step.



**VideoNote** Creating Scenes

## Creating Controls

The JavaFX library provides classes for all of the objects you can add to a GUI. The classes are in various subpackages of the `javafx` package, so you must write an `import` statement for the class you want to work with. For example, if you want to create a `Label` control, you will import the `Label` class, which is in the `javafx.scene.control` package.

Once you have written the necessary `import` statement, you will create an instance of the class. This is often done in the `start` method. For example, the following statement creates a `Label` control that displays the text *Hello World*:

```
Label messageLabel = new Label("Hello World");
```

Notice we pass the string we want displayed in the `Label` as an argument to the `Label` class's constructor. The `Label` class has multiple constructors, but this is the one you will use most often.

Regardless of the type of control you are creating, the process is typically the same: You import the class for the control, then in the `start` method you instantiate the class, calling the desired constructor.

## Creating Layout Containers

*Layout containers* are an important part of a JavaFX GUI. You use layout containers to arrange the positions of controls on the screen. JavaFX provides several different layout containers for you to choose from, and later in this chapter, we will look at many of them. For now, we will use only the three simple containers listed in [Table 11-1](#).

### Table 11-1 Three of the most simple layout containers

| Layout Container | Description |
|------------------|-------------|
|------------------|-------------|

|          |                                                    |
|----------|----------------------------------------------------|
| HBox     | Arranges controls in a single horizontal row.      |
| VBox     | Arranges controls in a single vertical row.        |
| GridPane | Arranges controls in a grid with rows and columns. |

[Figure 11-8](#) shows an example of how the three layout containers listed in [Table 11-1](#) affect the arrangement of controls. The leftmost screen shows an application with three Button controls in an HBox. The middle screen shows an application with three Button controls in a VBox. The rightmost screen shows an application with six Buttons in a GridPane.

## Figure 11-8 Examples of layout containers



[Figure 11-8 Full Alternative Text](#)

You create layout containers in the same manner that you create other GUI objects. First, you import the container's class from the necessary package. Then, you create an instance of the container's class, calling the desired constructor.

For example, if you want to create an HBox, you import the HBox class from the javafx.scene.layout package. Then, you create an instance of the HBox class. The following code snippet shows an example:

```
// Create a Label control.
Label messageLabel = new Label("Hello World");
```

```
// Create an HBox container and add the Label.
HBox hbox = new HBox(messageLabel);
```

First, this code creates a `Label` control. Then, it creates an `HBox` layout container, and adds the `Label` control to the `HBox`.

You can add multiple controls to an `HBox` by passing multiple arguments to the `HBox` constructor. For example, the following code snippet creates two `Label` controls, then adds them both to an `HBox`:

```
// Create two Label controls.
Label message1 = new Label("Hello");
Label message2 = new Label("World");
// Create an HBox container and add the Labels.
HBox hbox = new HBox(message1, message2);
```

This gives you a general idea of how to create a layout container, using `HBox` as an example. We will look at layout containers, including `HBox`, in greater detail as we progress through this chapter.

## Creating a Scene Object

Once you have created the controls that you want in your scene, and added them to a layout container, the next step is to create an instance of the `Scene` class (which is in the `javafx.scene` package), and add the layout container to the `Scene` object.

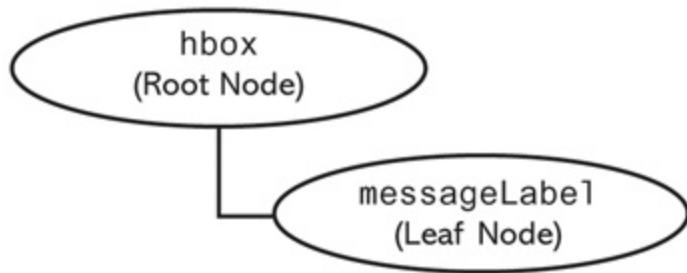
The following code snippet shows an example. It creates a `Label` control, adds the `Label` to an `HBox` container, then adds the `HBox` container to a `Scene` object:

```
// Create a Label control.
Label messageLabel = new Label("Hello World");
// Create an HBox container and add the Label.
HBox hbox = new HBox(messageLabel);
// Create a Scene and add the HBox as the root node.
Scene scene = new Scene(hbox);
```

The argument we pass to the `Scene` constructor is the scene's *root node*. In

this example, the `HBox` container will become the root node of the scene graph, as shown in [Figure 11-9](#).

## Figure 11-9 Nodes in the scene graph



[Figure 11-9 Full Alternative Text](#)

## Adding the Scene Object to the Stage

Once the `Scene` object has been created, the next step is to add the `Scene` object to the stage. Recall the `start` method receives a reference to a `Stage` object as an argument. To add your `Scene` object to the stage, call the `Stage` object's `setScene` method, passing a reference to your `Scene` object as an argument.

Let's assume `primaryStage` references the `Stage` object, and `scene` references your `Scene` object. Here is an example of how you would call the `setScene` method to add the `Scene` to the `Stage`:

```
primaryStage.setScene(scene);
```

Now that we've covered the basics, we can look at a complete program that creates a simple `Scene`. Look at [Code Listing 11-2](#), a simple Hello World

application. This application's scene contains an HBox with a Label displaying the text *Hello World*. When you compile and execute this program, you will see a window similar to the one shown in [Figure 11-10](#).

## Figure 11-10 Window displayed by the HelloWorld application



## Code Listing 11-2 (HelloWorld.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.control.Label;
6
7 /**
8 * A JavaFX Hello World application
9 */
10
11 public class HelloWorld extends Application
12 {
13 public static void main(String[] args)
14 {
15 // Launch the application.
16 launch(args);
17 }
18
19 @Override
20 public void start(Stage primaryStage)
21 {
22 // Create a Label control.
23 Label messageLabel = new Label("Hello World");
24
25 // Put the Label in an HBox.
```

```

26 HBox hbox = new HBox(messageLabel);
27
28 // Create a Scene with the HBox as its root node.
29 Scene scene = new Scene(hbox);
30
31 // Add the Scene to the Stage.
32 primaryStage.setScene(scene);
33
34 // Set the stage title.
35 primaryStage.setTitle("My First Scene");
36
37 // Show the window.
38 primaryStage.show();
39 }
40 }
```

Let's take a closer look at the program in [Code Listing 11-2](#):

- Lines 1 through 5 import the following JavaFX classes: Application, Stage, Scene, HBox, and Label.
- The declaration for the `HelloWorld` class begins in line 11. Notice it extends the Application class.
- The `main` method appears in lines 13 through 17. The `main` method does only one thing: it calls the `launch` method in line 16. The `launch` method, which is inherited from the Application class, does a number of setup operations, including the following:
  - It creates a `Stage` object that will be the application's window.
  - It calls the `start` method, passing a reference to the `Stage` object as an argument.
- The `start` method appears in lines 20 through 39. This is the entry point for the application. The `start` method is an abstract method in the Application class, and we must override it. Notice the method has a parameter named `primaryStage`. The `primaryStage` variable will reference the `Stage` object created by the `launch` method.
- Line 23 creates a `Label` control named `messageLabel` that displays the

```
 string "Hello World".
```

- Line 26 creates an `HBox` container named `hbox`, and adds the `messageLabel` control to it.
- Line 29 creates a `Scene` object named `scene`, and adds `hbox` as the scene's root node.
- Line 32 calls the `primaryStage` object's `setScene` method to set the scene to the stage.
- Line 35 calls the `primaryStage` object's `setTitle` method. This sets the text that will be displayed in the window's title bar.
- Line 38 calls the `primaryStage` object's `show` method. This displays the application's window.

## Setting the Size of the Scene

Notice in [Figure 11-10](#) that the window is very small. In fact, it is just large enough to display the `Label` control and the standard Windows buttons in the title bar. By default, when you create a `Scene` object, the scene will be just large enough to display the contents of the scene. This is known as the scene's *preferred size*.

If you wish, you can specify the size of the scene by passing width and height arguments to the `Scene` class constructor. For example, suppose we modify line 29 of [Code Listing 11-2](#) as follows:

```
Scene scene = new Scene(hbox, 300, 100);
```

The first argument, `hbox`, is the root node. The second argument, 300, is the scene's width, in pixels. The third argument, 100, is the scene's height, in pixels. As a result of adding these width and height arguments to the constructor call, the program will display a window similar to [Figure 11-11](#).

# Figure 11-11 Window displayed by the modified HelloWorld application



## Aligning Controls in an HBox Layout Container

Notice in [Figure 11-11](#) that the `Label` control is displayed in the top-left corner of the scene. That is the default alignment of an `HBox` layout container. You can change the alignment of an `HBox` by calling its `setAlignment` method. For example, assume `hbox` references an `HBox` object. The following statement causes all controls in the `hbox` container to be centered within the container:

```
hbox.setAlignment(Pos.CENTER);
```

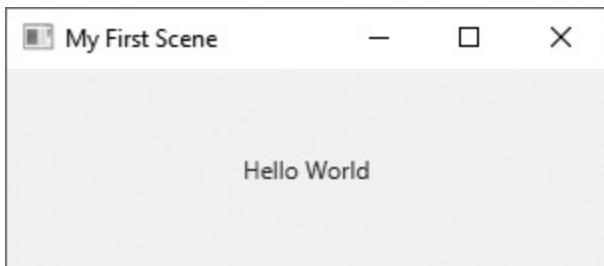
The argument you pass to the `setAlignment` method is one of the `enum` constants listed in [Table 11-2](#). The `Pos` type must be imported from the `javafx.geometry` package.

## Table 11-2 enum constants of the Pos type

|                              |                                |                               |
|------------------------------|--------------------------------|-------------------------------|
| <code>Pos.TOP_LEFT</code>    | <code>Pos.TOP_CENTER</code>    | <code>Pos.TOP_RIGHT</code>    |
| <code>Pos.CENTER_LEFT</code> | <code>Pos.CENTER</code>        | <code>Pos.CENTER_RIGHT</code> |
| <code>Pos.BOTTOM_LEFT</code> | <code>Pos.BOTTOM_CENTER</code> | <code>Pos.BOTTOM_RIGHT</code> |

For example, [Code Listing 11-3](#) shows a modified version of the `HelloWorld` program that you previously saw in [Code Listing 11-2](#). In this version, the `messageLabel` control is centered in an `HBox` that is 300 pixels wide and 100 pixels high. When you compile and execute this program, you will see a window similar to the one shown in [Figure 11-12](#).

## Figure 11-12 Window displayed by the `HelloWorld2` application



## Code Listing 11-3 (`HelloWorld2.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.control.Label;
6 import javafx.geometry.Pos;
7
8 /**
9 * A JavaFX Hello World application
10 */
11
12 public class HelloWorld2 extends Application
13 {
```

```

14 public static void main(String[] args)
15 {
16 // Launch the application.
17 launch(args);
18 }
19
20 @Override
21 public void start(Stage primaryStage)
22 {
23 // Create a Label control.
24 Label messageLabel = new Label("Hello World");
25
26 // Put the Label in an HBox.
27 HBox hbox = new HBox(messageLabel);
28
29 // Create a Scene with the HBox as its root node.
30 Scene scene = new Scene(hbox, 300, 100);
31
32 // Set the HBox's alignment to center.
33 hbox.setAlignment(Pos.CENTER);
34
35 // Add the Scene to the Stage.
36 primaryStage.setScene(scene);
37
38 // Set the stage title.
39 primaryStage.setTitle("My First Scene");
40
41 // Show the window.
42 primaryStage.show();
43 }
44 }
```

Notice in line 6, we imported the Pos enumeration from the javafx.geometry package. In line 30, we create a Scene object that is 300 pixels wide by 100 pixels high, with the hbox object as the root node. Then, in line 33, we call the hbox object's setAlignment method, passing Pos.CENTER as an argument.



## Checkpoint

- 11.15 What is the general difference between an HBox layout container, and a VBox layout container?

2. 11.16 If you want to arrange controls in a grid, with rows and columns, what layout container would you use?
3. 11.17 The `Scene` class is in what package?
4. 11.18 How do you change the alignment of an `HBox` container?
5. 11.19 What package is the `Pos` enumeration in?

# 11.4 Displaying Images

## Concept:

You use both the `Image` and `ImageView` classes to display an image.

Displaying an image in a JavaFX application is a two-step process: First, you load the image into memory, then second, you display the image. This requires two classes from the JavaFX library: `Image` and `ImageView`. Both of these classes are in the `javafx.scene.image` package.



### VideoNote Displaying Images

The `Image` class can load an image from the computer's local file system, or from an Internet location. The class supports the BMP, JPEG, GIF, and PNG file types. To load an image of any of these types, create an instance of the `Image` class, passing the constructor a string argument that specifies the file's name and location. Here is an example:

```
Image image = new Image("file:HotAirBalloon.jpg");
```

In this example, the string that we are passing to the constructor specifies a file named HotAirBalloon.jpg. The part of the string that reads `file:` is a protocol identifier indicating that the file is on the local computer. Because no path was given, it is assumed the file is in the same directory or folder as the program's executable `.class` file. If you want to load an image from a different location, you can specify a path. Here is an example:

```
Image image = new Image("file:C:\\\\Images\\\\HotAirBalloon.jpg");
```

Next, you create an instance of the `ImageView` class, passing a reference to

the `Image` object as an argument to the constructor. Here is an example:

```
ImageView imageView = new ImageView(image);
```

Once you've created an `ImageView` object, you can add it to a layout container. The program in [Code Listing 11-1](#) shows an example. [Figure 11-13](#) shows the program's output.

## Figure 11-13 The ImageDemo application



## Code Listing 11-4 (ImageDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.image.Image;
6 import javafx.scene.image.ImageView;
```

```

7 /**
8 * An Image Demo
9 */
10
11 public class ImageDemo extends Application
12 {
13 public static void main(String[] args)
14 {
15 // Launch the application.
16 launch(args);
17 }
18
19
20 @Override
21 public void start(Stage primaryStage)
22 {
23 // Create an Image object.
24 Image image = new Image("file:HotAirBalloon.jpg");
25
26 // Create an ImageView object.
27 ImageView imageView = new ImageView(image);
28
29 // Put the ImageView in an HBox.
30 HBox hbox = new HBox(imageView);
31
32 // Create a Scene with the HBox as its root node.
33 Scene scene = new Scene(hbox);
34
35 // Add the Scene to the Stage.
36 primaryStage.setScene(scene);
37
38 // Set the stage title.
39 primaryStage.setTitle("Hot Air Balloon");
40
41 // Show the window.
42 primaryStage.show();
43 }
44 }

```

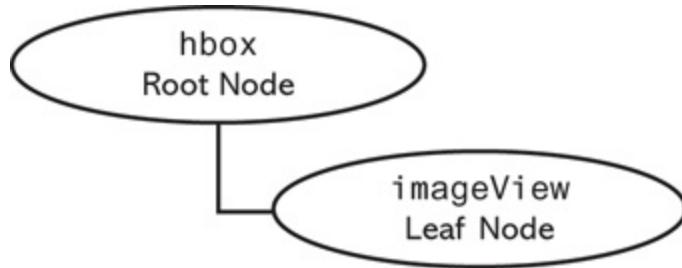
Let's take a closer look at the code:

- Lines 5 and 6 import the `Image` class and the `ImageView` class from the `javafx.scene.image` package.
- Line 24 creates an `Image` object that loads the file `HotAirBalloon.jpg` from the local file system.

- Line 27 creates an `ImageView` object, using the `Image` object created in line 24.
- Line 30 creates an `HBox`, adding the `ImageView` object to it.
- Line 33 creates a `Scene` object with the `HBox` as the root node.
- Line 36 sets the scene to the stage.
- Line 39 sets the text to be displayed in the window's title bar.
- Line 42 displays the application's window.

[Figure 11-14](#) depicts a scene graph for [Code Listing 11-4](#).

## Figure 11-14 Scene graph for the ImageDemo program in [Code Listing 11-4](#)



[Figure 11-14 Full Alternative Text](#)



### Note:

Notice in [Figure 11-14](#) that the `ImageView` object is a node in the scene graph, but the `Image` object is not. The `Image` class loads an image into memory, but it does not display the image on the screen. Displaying an image is the job of

the `ImageView` class. Therefore, the `ImageView` object is a node in the scene graph, but the `Image` object is not.

# Loading Images from an Internet Location

The string that you pass as an argument to the `Image` class constructor specifies a protocol used to load the image. The example you saw in [Code Listing 11-4](#) uses the `file:` protocol to load the image from the local file system. If you want to load an image from an Internet location, you can use the `http:` protocol, as shown in the following example:

```
Image image = new Image("http://www.gaddisbooks.com/images/
HotAirBalloon.jpg");
```

# Setting the Size of an Image

By default, the `ImageView` class displays an image at its full size. You can resize the image with the `ImageView` class's `setFitWidth` and `setFitHeight` methods. Let's assume `myImageView` references an `ImageView` object. The following code shows an example:

```
myImageView.setFitWidth(100);
myImageView.setFitHeight(100);
```

After this code executes, the image size of the `ImageView` object will be set to 100 pixels wide by 100 pixels high.



## Note:

By default, if you set the scene's size to a width and height that is different from the image's width and height, the image will *not* be scaled to fit the size

of the scene. If the image is smaller than the scene, the image will occupy only part of the window. If the image is larger than the program's window, only part of the image will be displayed.

## Preserving the Image's Aspect Ratio

Usually when you resize an image, you want to preserve the image's aspect ratio. The *aspect ratio* is the ratio of the image's width to the image's height. If you change an image's aspect ratio, the image will appear stretched (either horizontally or vertically).

The `ImageView` class has a method named `setPreserveRatio` that you can call to make sure an image's aspect ratio is preserved. Let's assume that `myImageView` references an `ImageView` object. The following code shows an example:

```
myImageView.setPreserveRatio(true);
```

The `setPreserveRatio` method accepts a Boolean argument. If you pass `true` as an argument, the image's aspect ratio is preserved as follows:

- If you set both the image's width and height, the width and/or the height may be scaled to get the closest possible fit while also preserving the aspect ratio.
- If you set only the image's width, the height will be scaled to preserve the aspect ratio.
- If you set only the image's height, the width will be scaled to preserve the aspect ratio.

If you pass `false` as an argument to the `setPreserveRatio` method, it is the same as not calling the method at all. As a result, any changes you make to the image's width and/or height will not preserve the image's aspect ratio. This will potentially cause the image to appear stretched in one direction.

# Changing an ImageView's Image

You can change the image that is displayed by an `ImageView` object by calling the object's `setImage` method. The `setImage` method takes, as an argument, a reference to an `Image` object you want to display. For example, let's assume `myImageView` references an `ImageView` object, and `myImage` references an `Image` object. The following code shows an example of calling the `setImage` method:

```
myImageView.setImageResource(myImage);
```

After this statement executes, the `myImageView` object will display the image that was loaded by the `myImage` object.



## Checkpoint

1. 11.20 What package contains the `Image` and `ImageView` classes?
2. 11.21 If you want to display an image in a layout container, do you add an `Image` object, or an `ImageView` object to the layout container?
3. 11.22 How do you set the width and height of an `ImageView`?
4. 11.23 If you want to make sure that an image does not appear stretched after it is resized, what method do you call?
5. 11.24 If you want to change the image that an `ImageView` object is displaying, what method do you call?

# **11.5 More about the HBox, VBox, and GridPane Layout Containers**

## **Concept:**

The layout containers provide methods for fine-tuning the way controls are arranged.

We have briefly introduced the `HBox`, `VBox`, and `GridPane` layout containers. In this section, we will look more closely at each of these layout containers, and discuss ways to fine-tune the ways in which they arrange controls.

## **The HBox Layout Container**

You have already seen programs that use the `HBox` layout container, which arranges controls in a single horizontal row. You can add one or more controls to an `HBox` by passing references to those controls as arguments to the `HBox` constructor. For example, the program in [Code Listing 11-5](#) displays three images in an `HBox`. The program's output is shown in [Figure 11-15](#).

## **Figure 11-15 The HBoxImages application**



[VideoNote](#) The HBox Layout Container

## Code Listing 11-5 (**HBoxImages.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.image.Image;
6 import javafx.scene.image.ImageView;
7
8 /**
9 * This program displays images in an Hbox.
10 */
11
12 public class HBoxImages extends Application
13 {
14 public static void main(String[] args)
15 {
16 // Launch the application.
17 launch(args);
18 }
19
20 @Override
21 public void start(Stage primaryStage)
22 {
```

```

23 // Create the Image objects.
24 Image moonImage = new Image("file:Moon.jpg");
25 Image shipImage = new Image("file:Ship.jpg");
26 Image sunsetImage = new Image("file:Sunset.jpg");
27
28 // Create the ImageView objects.
29 ImageView moonIView = new ImageView(moonImage);
30 ImageView shipIView = new ImageView(shipImage);
31 ImageView sunsetIView = new ImageView(sunsetImage);
32
33 // Resize the moon image, preserving its aspect ratio.
34 moonIView.setFitWidth(200);
35 moonIView.setPreserveRatio(true);
36
37 // Resize the ship image, preserving its aspect ratio.
38 shipIView.setFitWidth(200);
39 shipIView.setPreserveRatio(true);
40
41 // Resize the sunset image, preserving its aspect ratio.
42 sunsetIView.setFitWidth(200);
43 sunsetIView.setPreserveRatio(true);
44
45 // Put the ImageViews in an HBox.
46 HBox hbox = new HBox(moonIView, shipIView, sunsetIView);
47
48 // Create a Scene with the HBox as its root node.
49 Scene scene = new Scene(hbox);
50
51 // Add the Scene to the Stage.
52 primaryStage.setScene(scene);
53
54 // Set the stage title.
55 primaryStage.setTitle("Images");
56
57 // Show the window.
58 primaryStage.show();
59 }
60 }
```

Let's take a closer look at the code:

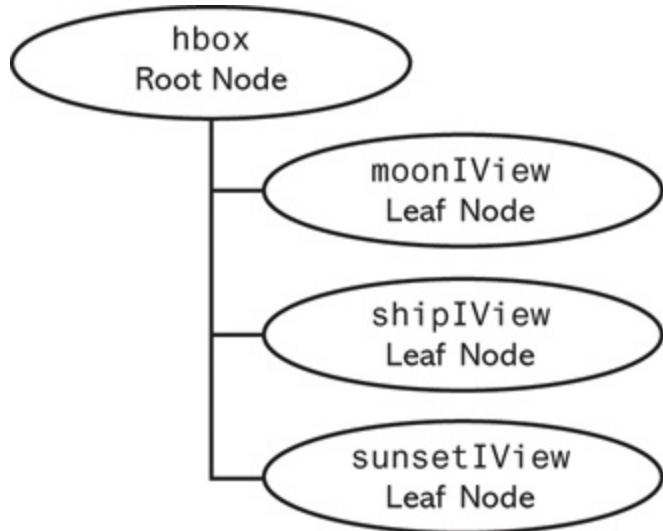
- Lines 24 through 26 create three `Image` objects, and load the image files `Moon.jpg`, `Ship.jpg`, and `Sunset.jpg`.
- Lines 29 through 31 create three `ImageView` objects for the image files that were loaded. The names of the `ImageFile` objects are `moonIView`,

`shipIView`, and `sunsetIView`.

- Line 34 sets the width of the `moonIView` object to 200 pixels. Line 35 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 38 sets the width of the `shipIView` object to 200 pixels. Line 39 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 42 sets the width of the `sunsetIView` object to 200 pixels. Line 43 causes the aspect ratio to be preserved. As a result, the image's height will be automatically scaled so the image does not appear stretched.
- Line 46 creates an `HBox` container, and adds the `moonIView`, `shipIView`, and `sunsetIView` objects to it.
- Line 49 creates a `Scene` object with the `HBox` as the root node.
- Line 52 sets the scene to the stage.
- Line 55 sets the text that will be displayed in the window's title bar.
- Line 58 displays the application's window.

[Figure 11-16](#) depicts a scene graph for [Code Listing 11-5](#).

## **Figure 11-16 Scene graph for the HBoxImages program in [Code Listing 11-5](#)**



[Figure 11-16 Full Alternative Text](#)

## Spacing

Notice in [Figure 11-15](#), the images are displayed with no space between them. If we want some space to appear between the controls in an `HBox`, we can optionally pass a spacing value as the first argument to the `HBox` constructor. Here is an example of how we might have written the statement in line 46 of [Code Listing 11-5](#):

```
HBox hbox = new HBox(10, moonIView, shipIView, sunsetIView);
```

If the first argument passed to the `HBox` constructor is a number, that value is used as the number of pixels to appear between the controls horizontally in the container. In this example, the images will be displayed with 10 pixels of space between them. This will cause the program's window to appear as shown in [Figure 11-17](#).

## Figure 11-17 Images in an `HBox` with 10 pixels spacing



# Padding

*Padding* is space that appears around the inside edge of a container. If you want to add padding to an `HBox`, you call the `HBox` object's `setPadding` method. The `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to an `HBox` container named `hbox`:

```
hbox.setPadding(new Insets(10));
```

The `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program. [Code Listing 11-6](#) shows an example. The program's output is shown in [Figure 11-18](#).

**Figure 11-18 The  
HBoxImagesWithPadding  
application**



## Code Listing 11-6 (**HBoxImagesWithPadding.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.geometry.Insets;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8
9 /**
10 * This program demonstrates padding and spacing.
11 */
12
13 public class HBoxImagesWithPadding extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Create the Image components.
```

```

25 Image moonImage = new Image("file:Moon.jpg");
26 Image shipImage = new Image("file:Ship.jpg");
27 Image sunsetImage = new Image("file:Sunset.jpg");
28
29 // Create the ImageView components.
30 ImageView moonIView = new ImageView(moonImage);
31 ImageView shipIView = new ImageView(shipImage);
32 ImageView sunsetIView = new ImageView(sunsetImage);
33
34 // Resize the moon image, preserving its aspect ratio.
35 moonIView.setFitWidth(200);
36 moonIView.setPreserveRatio(true);
37
38 // Resize the ship image, preserving its aspect ratio.
39 shipIView.setFitWidth(200);
40 shipIView.setPreserveRatio(true);
41
42 // Resize the sunset image, preserving its aspect ratio.
43 sunsetIView.setFitWidth(200);
44 sunsetIView.setPreserveRatio(true);
45
46 // Put the ImageViews in an HBox with 10 pixels spacing.
47 HBox hbox = new HBox(10, moonIView, shipIView, sunsetIView
48
49 // Put 30 pixels of padding around the HBox.
50 hbox.setPadding(new Insets(30));
51
52 // Create a Scene with the HBox as its root node.
53 Scene scene = new Scene(hbox);
54
55 // Add the Scene to the Stage.
56 primaryStage.setScene(scene);
57
58 // Set the stage title.
59 primaryStage.setTitle("Images");
60
61 // Show the window.
62 primaryStage.show();
63 }
64 }
```

This program is a modification of the program you saw in [Code Listing 11-5](#). Here are the differences:

- Line 5 imports the Insets class from the javafx.geometry class.

- Line 47 creates an `HBox` container with 10 pixels of horizontal spacing between the images.
- Line 50 sets the `HBox` container's padding to 30 pixels.

## The `VBox` Layout Container

The `VBox` layout container arranges controls in a single vertical row. If you want to create a `VBox`, you import the `VBox` class from the `javafx.scene.layout` package. Then you create an instance of the `VBox` class. The following code snippet shows an example:



### Video Note The `VBox` Layout Container

```
// Create a Label control.
Label messageLabel = new Label("Hello World");
// Create a VBox container and add the Label.
VBox vbox = new VBox(messageLabel);
```

First, this code creates a `Label` control. Then, it creates a `VBox` layout container, and adds the `Label` control to the `VBox`.

You can add multiple controls to a `VBox` by passing multiple arguments to the `VBox` constructor. For example, the following code snippet creates two `Label` controls, then adds them both to a `VBox`:

```
// Create two Label controls.
Label message1 = new Label("Hello");
Label message2 = new Label("World");

// Create a VBox container and add the Labels.
VBox vbox = new VBox(message1, message2);
```

By default, items appear in a `VBox` with no space between them. If you want some space to appear between the objects in a `VBox`, you can optionally pass a spacing value as the first argument to the `VBox` constructor. Here is an example:

```
VBox vbox = new VBox (10, message1, message2);
```

If the first argument passed to the `VBox` constructor is a number, that value is used as the number of pixels to appear horizontally between the controls in the container. In this example, the `Labels` will be displayed with 10 pixels of space between them.

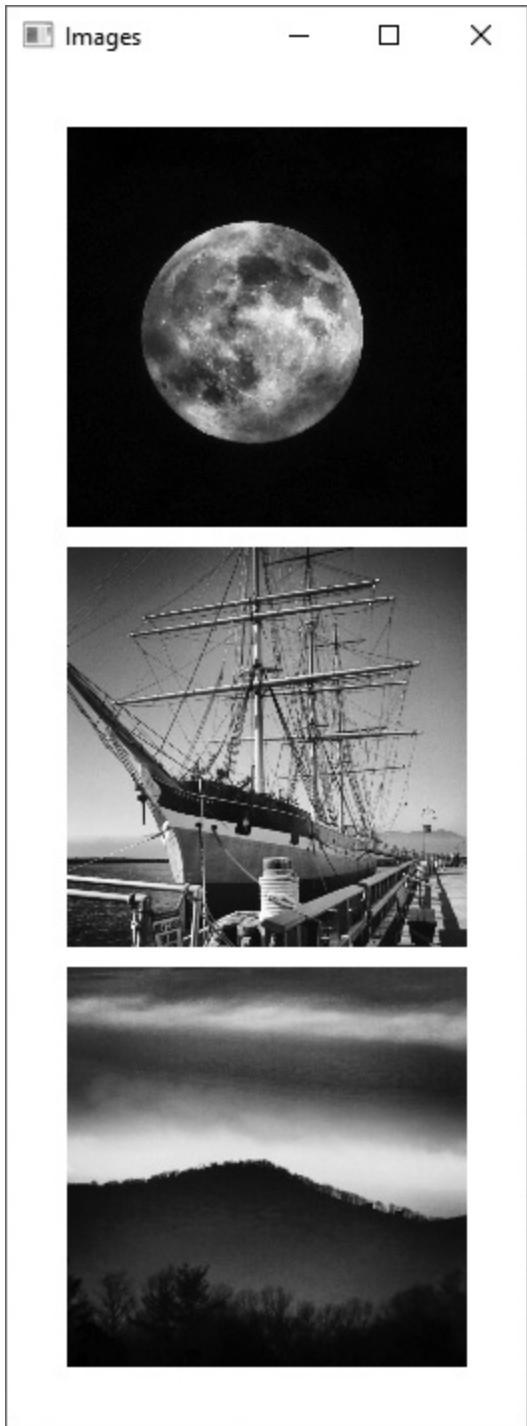
Recall that padding is space that appears around the inside edge of a container. By default, there is no padding in a `VBox`. If you want to add padding to a `VBox`, you call the `VBox` object's `setPadding` method. The `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to a `VBox` container named `vbox`:

```
vbox.setPadding(new Insets(10));
```

Recall that the `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program.

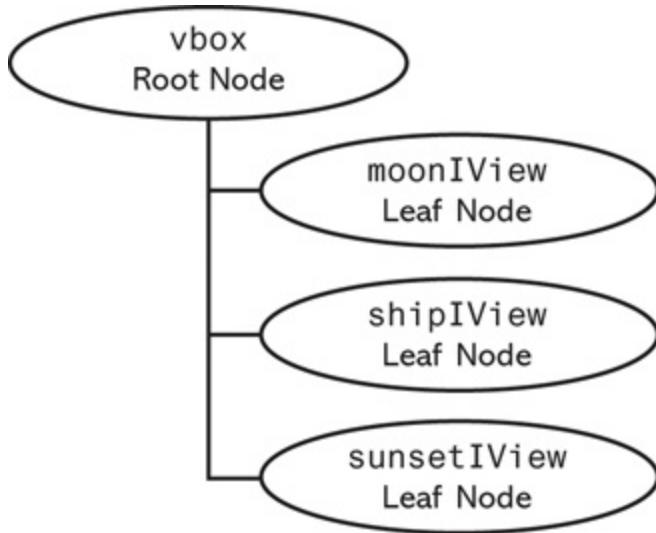
The program in [Code Listing 11-7](#) shows an example of how to use the `VBox` layout container. It is very similar to [Code Listing 11-6](#), except that it displays the three images in a `VBox`. The program's output is shown in [Figure 11-19](#). [Figure 11-20](#) depicts the application's scene graph.

## Figure 11-19 The VBoxImagesWithPadding application



**Figure 11-20 Scene graph for the VBoxImagesWithPadding**

# program in [Code Listing 11-7](#)



[Figure 11-20 Full Alternative Text](#)

## Code Listing 11-7 (**VBoxImagesWithPadding.java**)

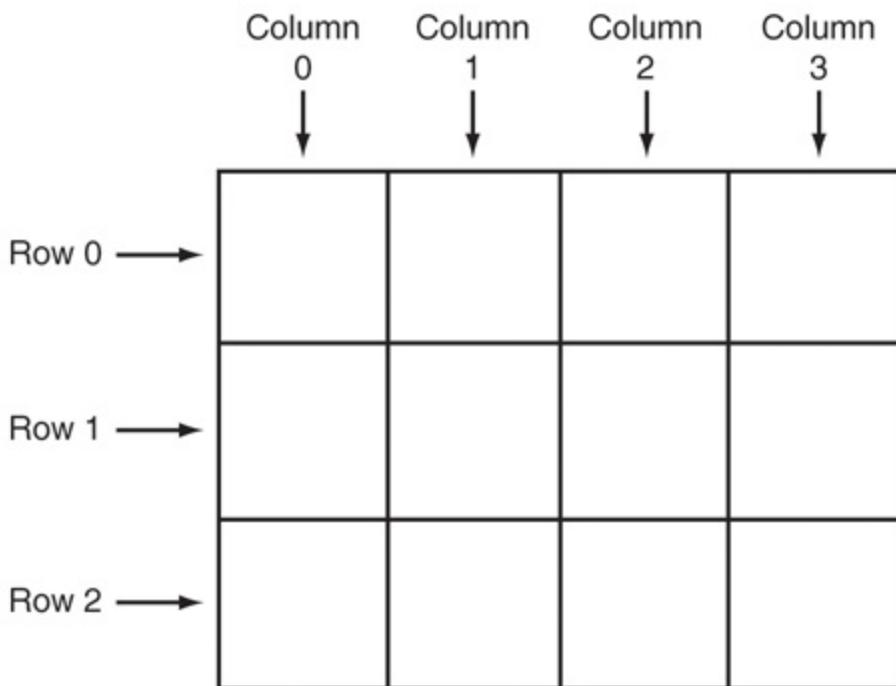
```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.geometry.Insets;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8
9 /**
10 * This program demonstrates the VBox layout container.
11 */
12
13 public class VBoxImagesWithPadding extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
```

```
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Create the Image components.
25 Image moonImage = new Image("file:Moon.jpg");
26 Image shipImage = new Image("file:Ship.jpg");
27 Image sunsetImage = new Image("file:Sunset.jpg");
28
29 // Create the ImageView components.
30 ImageView moonIView = new ImageView(moonImage);
31 ImageView shipIView = new ImageView(shipImage);
32 ImageView sunsetIView = new ImageView(sunsetImage);
33
34 // Resize the moon image, preserving its aspect ratio.
35 moonIView.setFitWidth(200);
36 moonIView.setPreserveRatio(true);
37
38 // Resize the ship image, preserving its aspect ratio.
39 shipIView.setFitWidth(200);
40 shipIView.setPreserveRatio(true);
41
42 // Resize the sunset image, preserving its aspect ratio.
43 sunsetIView.setFitWidth(200);
44 sunsetIView.setPreserveRatio(true);
45
46 // Put the ImageViews in a VBox with 10 pixels spacing.
47 VBox vbox = new VBox(10, moonIView, shipIView, sunsetIView);
48
49 // Put 30 pixels of padding around the VBox.
50 vbox.setPadding(new Insets(30));
51
52 // Create a Scene with the VBox as its root node.
53 Scene scene = new Scene(vbox);
54
55 // Add the Scene to the Stage.
56 primaryStage.setScene(scene);
57
58 // Set the stage title.
59 primaryStage.setTitle("Images");
60
61 // Show the window.
62 primaryStage.show();
63 }
64 }
```

# The GridPane Layout Container

The `GridPane` layout container arranges its contents in a grid with columns and rows. As a result, a `GridPane` is divided into cells, much like a spreadsheet. As shown in [Figure 11-21](#), the columns and rows are identified by indexes, beginning at 0. You use these column and row indexes to refer to specific cells within the `GridPane`.

## Figure 11-21 Column and row indexes



[VideoNote](#) The `GridPane` Layout Container

If you want to create a `GridPane`, you import the `GridPane` class from the `javafx.scene.layout` package. Then, you create an instance of the

`GridPane` class using its no-arg constructor, as shown here:

```
GridPane gridpane = new GridPane();
```

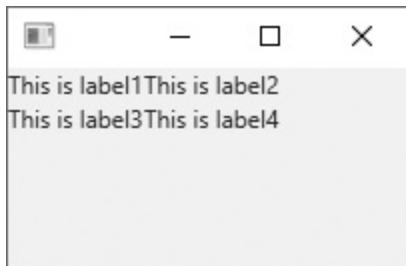
Once you have created a `GridPane` object, you can call the object's `add` method to add controls at specific positions in the grid. Here is the general format of the `add` method:

```
gridPaneObject.add(control, column, row);
```

In the general format, `gridPaneObject` is a reference to a `GridPane`, `control` is the control you are adding to the `GridPane`, `column` is the column index, and `row` is the row index.

The program in [Code Listing 11-8](#) demonstrates how to use a `GridPane`. The program's output is shown in [Figure 11-22](#).

## Figure 11-22 The GridPaneDemo application



[Figure 11-22 Full Alternative Text](#)

## Code Listing 11-8 (`GridPaneDemo.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
```

```

4 import javafx.scene.layout.GridPane;
5 import javafx.geometry.Insets;
6 import javafx.scene.control.Label;
7
8 /**
9 * GridPane Demo
10 */
11
12 public class GridPaneDemo extends Application
13 {
14 public static void main(String[] args)
15 {
16 // Launch the application.
17 launch(args);
18 }
19
20 @Override
21 public void start(Stage primaryStage)
22 {
23 // Create some Label controls.
24 Label label1 = new Label("This is label1");
25 Label label2 = new Label("This is label2");
26 Label label3 = new Label("This is label3");
27 Label label4 = new Label("This is label4");
28
29 // Create a GridPane.
30 GridPane gridpane = new GridPane();
31
32 // Add the Labels to the GridPane.
33 gridpane.add(label1, 0, 0); // Column 0, Row 0
34 gridpane.add(label2, 1, 0); // Column 1, Row 0
35 gridpane.add(label3, 0, 1); // Column 0, Row 1
36 gridpane.add(label4, 1, 1); // Column 1, Row 1
37
38 // Create a Scene with the GridPane as its root node.
39 // The Scene is 200 pixels wide by 100 pixels high.
40 Scene scene = new Scene(gridpane, 200, 100);
41
42 // Add the Scene to the Stage.
43 primaryStage.setScene(scene);
44
45 // Show the window.
46 primaryStage.show();
47 }
48 }

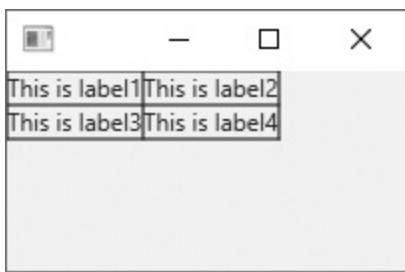
```

The `GridPane` class has a method named `setGridLinesVisible` that is

helpful for debugging purposes. When you call the method, passing `true` as its argument, the `GridPane` is displayed with grid lines. For example, if we added the following statement to [Code Listing 11-8](#), the program's output would appear as shown in [Figure 11-23](#):

```
gridpane.setGridLinesVisible(true);
```

## Figure 11-23 A GridPane with visible grid lines



[Figure 11-23 Full Alternative Text](#)

By default, there is no space between the columns and rows in a `GridPane`. If you want space to appear between the columns in a `GridPane`, you can call the `GridPane` class's `setHgap` method. The argument you pass to the `setHgap` method is the number of pixels of space you want between the columns. Let's assume `gridpane` references a `GridPane` object. The following statement shows an example of calling the `setHgap` method:

```
gridpane.setHgap(10);
```

This statement will cause 10 pixels of space to appear between the columns of the `GridPane`. If you want vertical space to appear between the rows in a `GridPane`, you can call the `GridPane` class's `setVgap` method. Once again, assume `gridpane` references a `GridPane` object. The following statement calls the `setHgap` method to display 10 pixels of space between the rows of the `GridPane` object:

```
gridpane.setVgap(10);
```

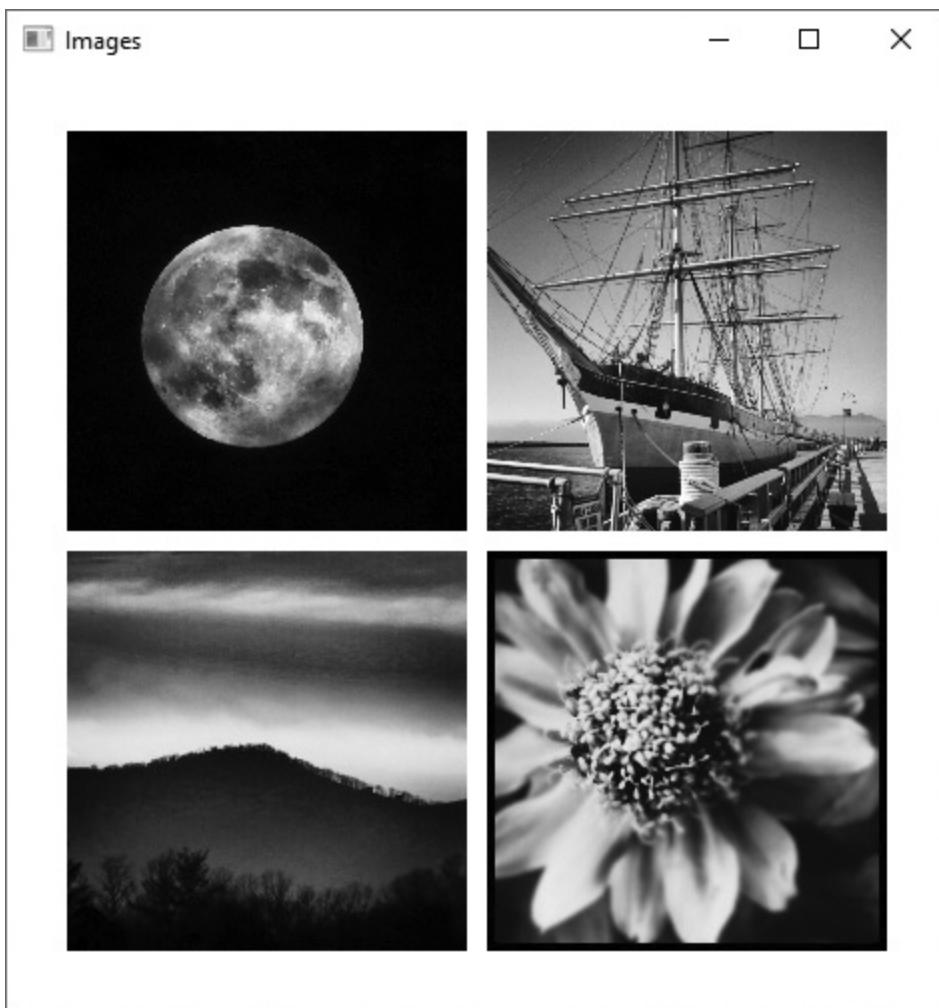
By default, there is no padding around the inside edge of a `GridPane`. However, you can add padding by calling the `GridPane` object's `setPadding` method. Recall that the `setPadding` method takes an `Insets` object as its argument. When constructing the `Insets` object, pass a number of pixels as an argument to the constructor. For example, the following statement adds 10 pixels of padding to a `GridPane` container named `gridpane`:

```
gridpane.setPadding(new Insets(10));
```

Recall that the `Insets` class is in the `javafx.geometry` package, so be sure to include the necessary `import` statement in your program.

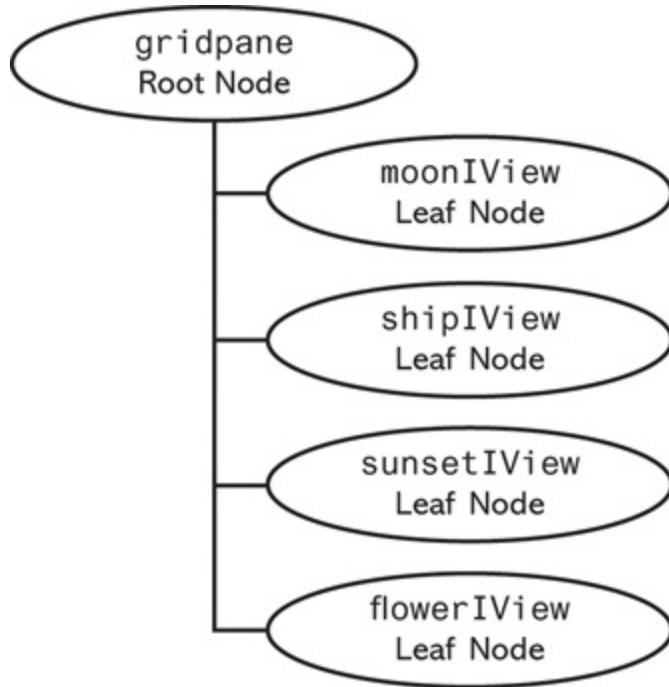
[Code Listing 11-9](#) shows a program that displays images in a `GridPane` with 4 columns and 2 rows. The `GridPane` has 10 pixels of space between the cells, and 30 pixels of padding. The program's output is shown in [Figure 11-24](#). [Figure 11-25](#) depicts the application's scene graph.

## Figure 11-24 The GridPaneImages application



[Figure 11-24 Full Alternative Text](#)

**Figure 11-25 Scene graph for the GridPaneImages program in [Code Listing 11-9](#)**



[Figure 11-25 Full Alternative Text](#)

## Code Listing 11-9 (**GridPaneImages.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.GridPane;
5 import javafx.geometry.Insets;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8
9 /**
10 * This program demonstrates the GridPane layout container.
11 */
12
13 public class GridPaneImages extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
```

```
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Create the Image components.
25 Image moonImage = new Image("file:Moon.jpg");
26 Image shipImage = new Image("file:Ship.jpg");
27 Image sunsetImage = new Image("file:Sunset.jpg");
28 Image flowerImage = new Image("file:Flower.jpg");
29
30 // Create the ImageView components.
31 ImageView moonIView = new ImageView(moonImage);
32 ImageView shipIView = new ImageView(shipImage);
33 ImageView sunsetIView = new ImageView(sunsetImage);
34 ImageView flowerIView = new ImageView(flowerImage);
35
36 // Resize the moon image, preserving its aspect ratio.
37 moonIView.setFitWidth(200);
38 moonIView.setPreserveRatio(true);
39
40 // Resize the ship image, preserving its aspect ratio.
41 shipIView.setFitWidth(200);
42 shipIView.setPreserveRatio(true);
43
44 // Resize the sunset image, preserving its aspect ratio.
45 sunsetIView.setFitWidth(200);
46 sunsetIView.setPreserveRatio(true);
47
48 // Resize the flower image, preserving its aspect ratio.
49 flowerIView.setFitWidth(200);
50 flowerIView.setPreserveRatio(true);
51
52 // Create a GridPane.
53 GridPane gridpane = new GridPane();
54
55 // Add the ImageViews to the GridPane.
56 gridpane.add(moonIView, 0, 0); // Col 0, Row 0
57 gridpane.add(shipIView, 1, 0); // Col 1, Row 0
58 gridpane.add(sunsetIView, 0, 1); // Col 0, Row 1
59 gridpane.add(flowerIView, 1, 1); // Col 1, Row 1
60
61 // Set the gap sizes.
62 gridpane.setVgap(10);
63 gridpane.setHgap(10);
64
65 // Set the GridPane's padding.
66 gridpane.setPadding(new Insets(30));
```

```

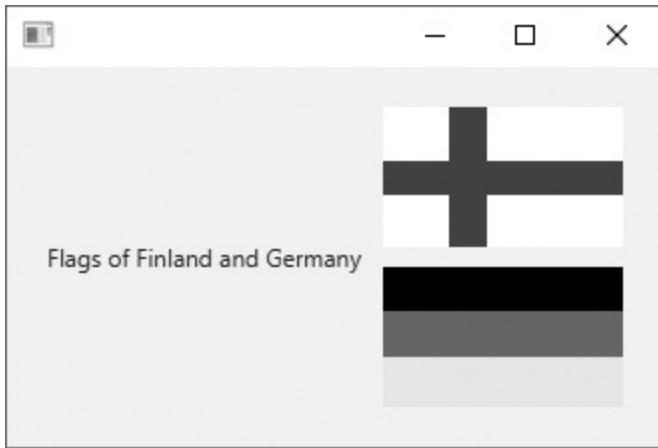
67
68 // Create a Scene with the GridPane as its root node.
69 Scene scene = new Scene(gridpane);
70
71 // Add the Scene to the Stage.
72 primaryStage.setScene(scene);
73
74 // Set the stage title.
75 primaryStage.setTitle("Images");
76
77 // Show the window.
78 primaryStage.show();
79 }
80 }
```

## Using Multiple Layout Containers in the Same Screen

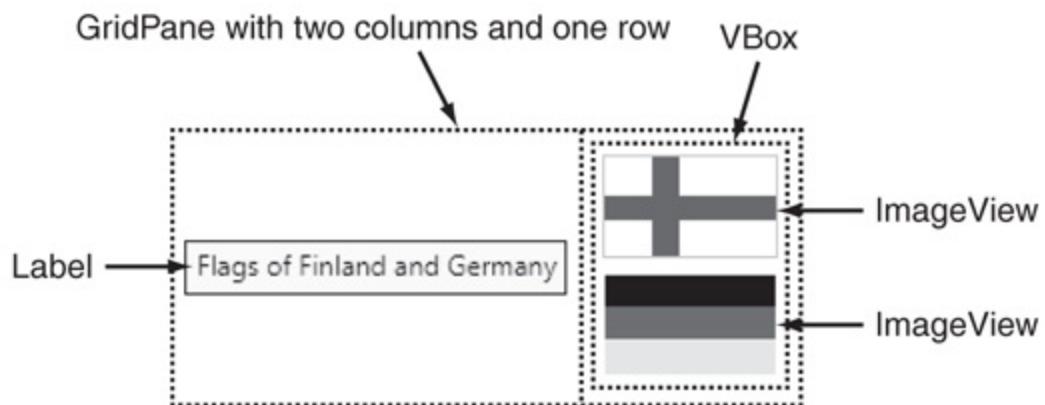
Quite often, to get the particular screen layout that you desire, you will have to use multiple layout containers. You can even nest a layout container inside of another layout container. For example, [Figure 11-26](#) shows a window that has a `GridPane` container with a `VBox` container nested inside of it. Here are some details about the containers in the figure (as illustrated in [Figure 11-27](#)):

- The `GridPane` has two columns and one row.
- The `Label` control is in the `GridPane`'s left column, and the `VBox` is in the `GridPane`'s right column.
- The `ImageView` objects are in the `GridPane`.

## Figure 11-26 A window with nested layout containers



## Figure 11-27 The layout



[Figure 11-27 Full Alternative Text](#)



## Checkpoint

- 11.25 What is the difference between spacing and padding, in regard to layout containers?
- 11.26 What package is the `Insets` class in?
- 11.27 How do you set the horizontal and vertical spacing in a `GridPane` container?

4. 11.28 Once you have created a `GridPane` container, how do you add a control to a specific column and row?

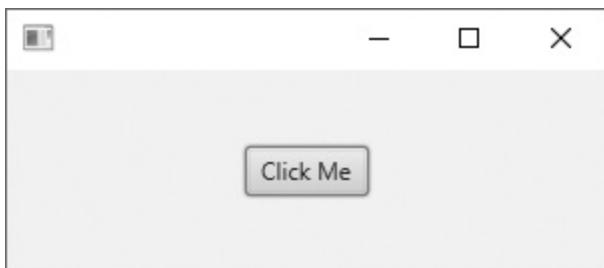
# 11.6 Button Controls and Events

## Concept:

A **Button** is a control the user can click to cause an action that you have programmed to take place.

A *Button* is a rectangular control that appears as a button with a caption written across its face. Typically, when the user clicks a **Button** control (either with a mouse, or by touching it on a touch screen), an action takes place. [Figure 11-28](#) shows an example of a JavaFX window that contains a **Button** control.

## Figure 11-28 A window with a Button control



**VideoNote** Button Controls and Events

To create a **Button** control, you will use the **Button** class, which is in the `javafx.scene.control` package. Once you have written the necessary import statement, you will create an instance of the **Button** class. The

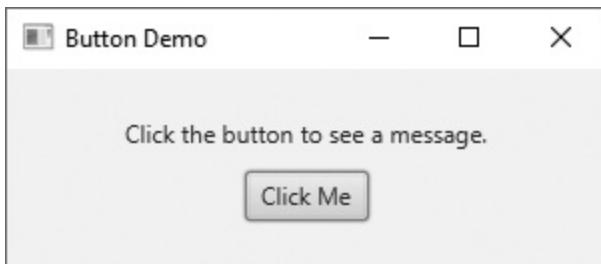
following statement shows an example of creating a `Button` object that displays the text *Click Me*:

```
Button myButton = new Button("Click Me");
```

Notice that we pass the string we want displayed on the `Button` as an argument to the `Button` class's constructor. The `Button` class has multiple constructors, but this is the one you will use most often.

For example, look at [Code Listing 11-10](#). When you compile and execute this program, you will see the window shown in [Figure 11-29](#).

## Figure 11-29 The ButtonDemo application



## Code Listing 11-10 (ButtonDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.geometry.Pos;
8
9 /**
10 * A Button Demo
11 */
```

```

12
13 public class ButtonDemo extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Create a Label control.
25 Label myLabel = new Label("Click the button to see a messa
26
27 // Create a Button control.
28 Button myButton = new Button("Click Me");
29
30 // Put the Label and Button in a VBox with 10 pixels of sp
31 VBox vbox = new VBox(10, myLabel, myButton);
32
33 // Create a Scene with the VBox as its root node.
34 Scene scene = new Scene(vbox, 300, 100);
35
36 // Set the scene's alignment to center.
37 vbox.setAlignment(Pos.CENTER);
38
39 // Add the Scene to the Stage.
40 primaryStage.setScene(scene);
41
42 // Set the stage title.
43 primaryStage.setTitle("Button Demo");
44
45 // Show the window.
46 primaryStage.show();
47 }
48 }
```

Here are some specific things to note about the `ButtonDemo` program:

- Line 6 imports the `Button` class from the `javafx.control` package.
- Line 28 creates an instance of the `Button` class. The resulting `Button` object is referenced by the `myButton` variable. The string "Click Me" is displayed on the `Button`.

If you compile and run this application, you will notice nothing happens when you click the Button. That requires some extra code. We will discuss that next, but for now, you can end the application by clicking the standard Close button in the window's title bar.

## Handling Events

An *event* is an action that takes place while a program is running. For example, each time the user clicks a `Button` control, an event takes place. When an event takes place, the control responsible for the event creates an *event object* that contains information about the event. The GUI control that created the event object is known as the *event source*.

Event objects are instances of the `Event` class (from the `javafx.event` package), or one of its subclasses. When a `Button` control is clicked, an event object that is an instance of the `ActionEvent` class is created. `ActionEvent` is a subclass of the `Event` class, and it is also in the `javafx.event` package.

But what happens to the event object once it is generated by an event source? It's possible the event source is connected to one or more event handlers. An *event handler* is an object that responds to events. If an event source is connected to an event handler, a specific method in the event handler is called and the event object is passed as an argument to the method. This process is sometimes referred to as *event firing*.

## Writing Event Handlers

When you are writing a GUI application, it is your responsibility to write the event handler classes that your application needs. When you write an event handler class, it must implement the `EventHandler` interface, which is in the `javafx.event` package. The `EventHandler` interface specifies a `void` method named `handle`. The `handle` method must have a parameter of the `Event` class, or one of its subclasses.

For example, suppose we want to write an event handler class that can

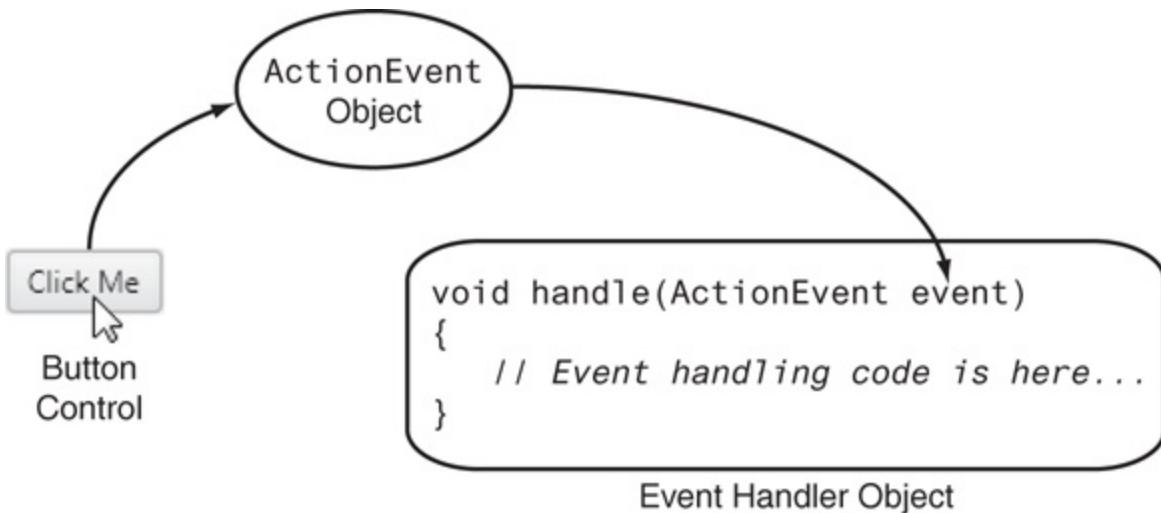
respond to Button clicks. Recall that when a Button control is clicked, the event object that is generated is of the ActionEvent type. This is the general format of an event handler class that handles events of the ActionEvent type:

```
class ButtonClickHandler implements EventHandler<ActionEvent>
{
 @Override
 public void handle(ActionEvent event)
 {
 // Write event handling code here.
 }
}
```

Notice the first line of the class declaration ends with the clause `implementsEventHandler <ActionEvent>`. This specifies that the class implements the `EventHandler` interface, and the type of event object that it will work with is `ActionEvent`. Next, notice that the `handle` method has an `ActionEvent` parameter named `event`. When the `handle` method is called, an `ActionEvent` object will be passed to it as an argument. Any code you want to execute as the result of a Button click must be written in the `handle` method.

Once you have written an event handler class, you create an object of that class, and connect the event handler object with a control. When a Button control is clicked, it generates an `ActionEvent` object, and it automatically executes the `handle` method of the event handler object to which it is connected, passing the `ActionEvent` object as an argument. This is illustrated in [Figure 11-30](#).

## Figure 11-30 A Button control firing an ActionEvent



[Figure 11-30 Full Alternative Text](#)

## Registering an Event Handler

Once you have written an event handler class, you create an object of the class, and connect the object to a control. The process of connecting an event handler object to a control is called *registering* the event handler. Button controls have a method named `setOnAction`, which is used for registering event handlers. You simply call the Button's `setOnAction` method, passing a reference to the event handler object as an argument. This will connect the Button control to the event handler object.

For example, suppose we create a Button control named `myButton`, and we have written an event handler class named `ButtonClickHandler`. The following code registers an object of the `ButtonClickHandler` class with the Button control:

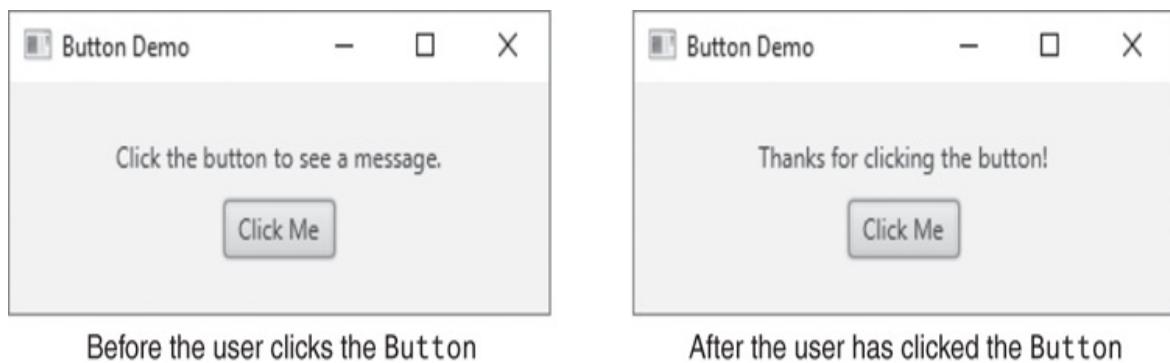
```
myButton.setOnAction(new ButtonClickHandler());
```

Now, when the user clicks on the Button control, the `ButtonClickHandler` object's `handle` method will be automatically executed.

A common technique for writing an event listener class is to write it as a private inner class inside the class that creates the GUI. [Code Listing 11-11](#) shows an example. The program's output is shown in [Figure 11-31](#). The

image on the left shows the application when it starts, and the image on the right shows the application after the user has clicked the Button.

## Figure 11-31 The EventDemo application



## Code Listing 11-11 (EventDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.geometry.Pos;
8 import javafx.event.EventHandler;
9 import javafx.event.ActionEvent;
10
11 /**
12 * An Event Demo
13 */
14
15 public class EventDemo extends Application
16 {
17 // Field for the Label control
18 private Label myLabel;
19 }
```

```
20 public static void main(String[] args)
21 {
22 // Launch the application.
23 launch(args);
24 }
25
26 @Override
27 public void start(Stage primaryStage)
28 {
29 // Create a Label control.
30 myLabel = new Label("Click the button to see a message.");
31
32 // Create a Button control.
33 myButton = new Button("Click Me");
34
35 // Register the event handler.
36 myButton.setOnAction(new ButtonClickHandler());
37
38 // Put the Label and Button in a VBox with 10 pixels of sp
39 VBox vbox = new VBox(10, myLabel, myButton);
40
41 // Create a Scene with the VBox as its root node.
42 Scene scene = new Scene(vbox, 300, 100);
43
44 // Set the scene's alignment to center.
45 vbox.setAlignment(Pos.CENTER);
46
47 // Add the Scene to the Stage.
48 primaryStage.setScene(scene);
49
50 // Set the stage title.
51 primaryStage.setTitle("Button Demo");
52
53 // Show the window.
54 primaryStage.show();
55 }
56
57 /**
58 * Event handler class for myButton
59 */
60
61 class ButtonClickHandler implements EventHandler<ActionEvent>
62 {
63 @Override
64 public void handle(ActionEvent event)
65 {
66 myLabel.setText("Thanks for clicking the button!");
67 }
68 }
```

```
67 }
68 }
69 }
```

This application has a `Label` control, named `myLabel`, that initially displays *Click the button to see a message*. When the user clicks the `Button` control, the text that is displayed by `myLabel` changes to *Thanks for clicking the button!* Let's take a closer look at the program to see how it works:

- Line 8 imports the `javafx.event.EventHandler` class, and line 9 imports the `javafx.event.ActionEvent` class. These classes are needed by our event handler class.
- Line 18 declares `myLabel` as a field in the `EventDemo` class. The reason we are declaring it as a field (instead of declaring it as a local variable in the `start` method) is that the event handler object needs to access it.
- Inside the `start` method, in line 30, we create a `Label` control and assign it to the `myLabel` field.
- In line 33, we create a `Button` control named `myButton`.
- Line 36 calls the `myButton` control's `setOnAction` method to register an event handler with the `Button`. To accomplish this, the code in this line does two things: it creates an instance of the `ButtonClickHandler` class (which we will see in a moment), and it passes a reference to that object as an argument to the `setOnAction` method.
- Lines 39 through 54 perform all the necessary steps to place the `Label` and the `Button` in a `VBox`, add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 61 through 68. It is an inner class named `ButtonClickHandler`. The `handle` method appears in lines 64 through 67. Notice the `handle` method has one parameter, `event`, which is an `ActionEvent` object. This parameter receives the event object that is passed to the method when it is called. Although we do not actually use the `event` parameter in this method, we still have to list it inside the

method header's parentheses, because it is required by the `ActionListener` interface.

- The statement in line 66 calls the `myLabel` object's `setText` method to change the text that the `Label` displays.



## Checkpoint

1. 11.29 What is an event? Give an example.
2. 11.30 What is an event handler?
3. 11.31 In what package is the `EventHandler` interface?
4. 11.32 What method do you call to register an event handler with a `Button` control?

# 11.7 Reading Input with `TextField` Controls

## Concept:

The `TextField` control is a rectangular area that can accept keyboard input from the user.

One of the primary controls that you will use to get data from the user is the `TextField` control. A `TextField` control appears as a rectangular area. When the application is running, the user can type text into a `TextField` control. In the program, you can retrieve the text that the user entered and use that text in any necessary operations.



### Video Note The `TextField` Control

To create a `TextField` control, you will use the `TextField` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `TextField` class. The following statement shows an example:

```
TextField myTextField = new TextField();
```

This statement creates an empty `TextField`. Optionally, you can pass a string to the constructor to display initial text in the `TextField`. Here is an example:

```
TextField myTextField = new TextField("Example data");
```

To retrieve the text that the user has typed into a `TextField` control, you call the control's `getText` method. The method returns the value that has been

entered into the `TextField` as a `String`. For example, assume `myTextField` is a `TextField` control. The following code reads the value that has been entered into `myTextField`, and assigns it to a `String` variable named `input`:

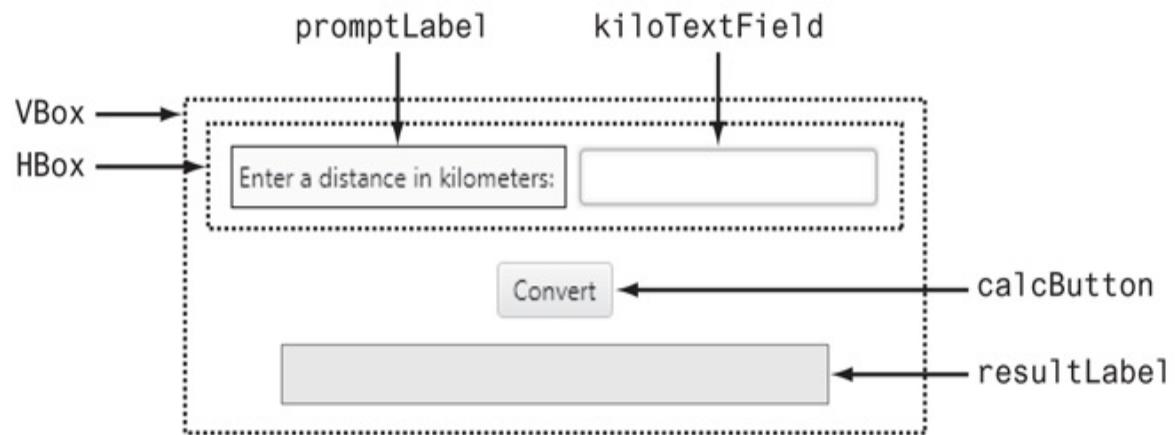
```
String input;
input = myTextField.getText();
```

To demonstrate the `TextField` control, we will look at the Kilometer Converter application in [Code Listing 11-12](#). This application presents a window in which the user can enter a distance in kilometers, then click a button to see that distance converted to miles. The conversion formula is:

$$\text{Miles} = \text{Kilometers} \times 0.6214$$

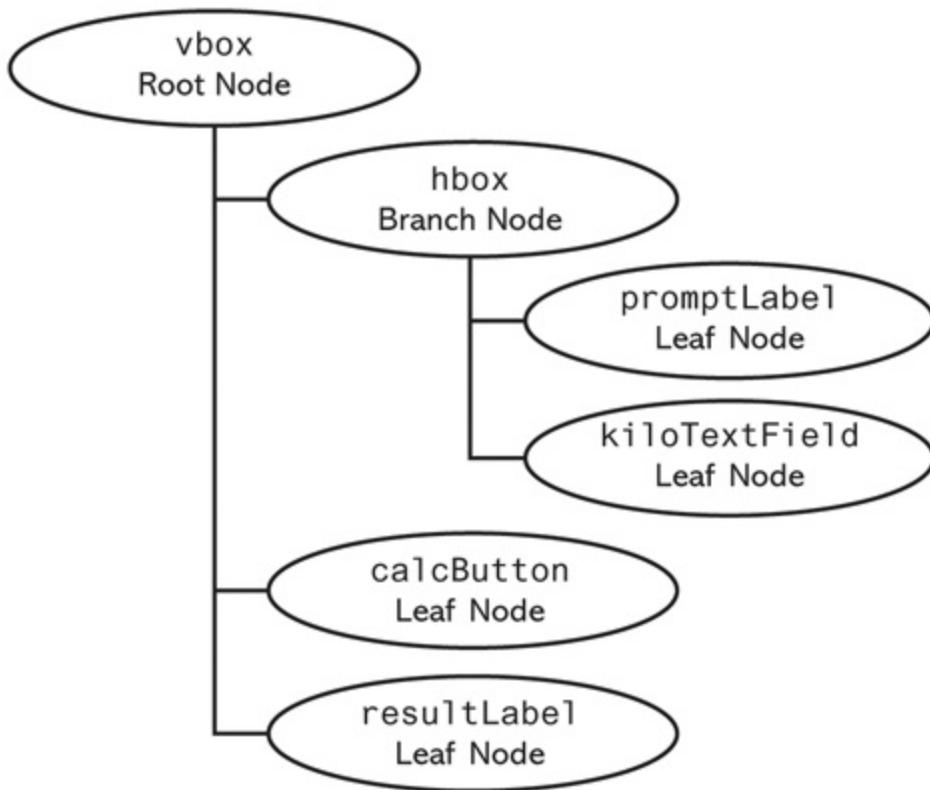
[Figure 11-32](#) shows the layout of the controls in the Kilometer Converter application. Notice we use two layout containers: an `HBox` nested inside of a `VBox`. The `HBox` contains the `promptLabel` control and the `kiloTextField` control. The `VBox`, which is the root node, contains the `HBox`, the `calcButton` control, and the `resultLabel` control. [Figure 11-33](#) depicts the scene graph for the application.

## **Figure 11-32 Layout of the controls in the Kilometer Converter application**



[Figure 11-32 Full Alternative Text](#)

## Figure 11-33 Scene graph for the Kilometer Converter application



[Figure 11-33 Full Alternative Text](#)

[Code Listing 11-12](#) shows the application's code. The program's output is shown in [Figure 11-34](#). The image on the left shows the application when it starts, and the image on the right shows the application after the user has entered 100 into the `kiloTextField` control, then clicked the `calcButton` control. Notice the conversion value is displayed in the `resultLabel` control.

## Figure 11-34 The Kilometer Converter application



[Figure 11-34 Full Alternative Text](#)

## Code Listing 11-12 (**KiloConverter.java**)

```

1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextField;
10 import javafx.scene.control.Button;
11 import javafx.event.EventHandler;
12 import javafx.event.ActionEvent;
13
14 /**
15 * Kilometer Converter application
16 */
17
18 public class KiloConverter extends Application
19 {
20 // Fields
21 private TextField kiloTextField;
22 private Label resultLabel;
23
24 public static void main(String[] args)
25 {
26 // Launch the application.

```

```
27 launch(args);
28 }
29
30 @Override
31 public void start(Stage primaryStage)
32 {
33 // Create a Label to display a prompt.
34 Label promptLabel = new Label("Enter a distance in kilomet
35
36 // Create a TextField for input.
37 kiloTextField = new TextField();
38
39 // Create a Button to perform the conversion.
40 Button calcButton = new Button("Convert");
41
42 // Register the event handler.
43 calcButton.setOnAction(new CalcButtonHandler());
44
45 // Create an empty Label to display the result.
46 resultLabel = new Label();
47
48 // Put the promptLabel and the kiloTextField in an HBox.
49 HBox hbox = new HBox(10, promptLabel, kiloTextField);
50
51 // Put the HBox, calcButton, and resultLabel in a VBox.
52 VBox vbox = new VBox(10, hbox, calcButton, resultLabel);
53
54 // Set the VBox's alignment to center.
55 vbox.setAlignment(Pos.CENTER);
56
57 // Set the VBox's padding to 10 pixels.
58 vbox.setPadding(new Insets(10));
59
60 // Create a Scene.
61 Scene scene = new Scene(vbox);
62
63 // Add the Scene to the Stage.
64 primaryStage.setScene(scene);
65
66 // Set the stage title.
67 primaryStage.setTitle("Kilometer Converter");
68
69 // Show the window.
70 primaryStage.show();
71 }
72
73 /*
74 * Event handler class for calcButton
```

```

75 */
76
77 class CalcButtonHandler implements EventHandler<ActionEvent>
78 {
79 @Override
80 public void handle(ActionEvent event)
81 {
82 // Get the kilometers.
83 double kilometers = Double.parseDouble(kiloTextField.getText());
84
85 // Convert the kilometers to miles.
86 double miles = kilometers * 0.6214;
87
88 // Display the results.
89 resultLabel.setText(String.format("%,.2f miles", miles));
90 }
91 }
92 }
```

Let's take a closer look at the application:

- Lines 1 through 12 have the necessary JavaFX import statements.
- Lines 21 and 22 declare `kiloTextField` and `resultLabel` as fields in the `KiloConverter` class. The reason we are declaring them as fields (instead of declaring them as local variables in the `start` method) is that the event handler object needs to access them.
- Line 34 creates a `Label` control named `promptLabel`, to prompt the user to enter a distance in kilometers.
- Line 37 creates a `TextField` control and assigns it to the `kiloTextField` field.
- Line 40 creates a `Button` control named `calcButton`.
- Line 43 calls the `calcButton` control's `setOnAction` method to register an event handler with the `Button`. To accomplish this, the code in this line does two things: it creates an instance of the `CalcButtonHandler` class (which we will see in a moment), and it passes a reference to that object as an argument to the `setOnAction` method.

- Line 46 creates a `Label` control and assigns it to the `resultLabel` field. Notice we did not pass a string argument to the `Label` class constructor. As a result, the `Label` will be empty (it will not initially display any text).
- Line 49 puts the `promptLabel` and `kiloTextField` controls in an `HBox`, with 10 pixels of spacing between them.
- Line 52 puts the `HBox`, the `calcButton` control, and the `resultLabel` control in a `VBox`, with 10 pixels of spacing between them.
- Lines 55 and 58 set the `VBox`'s alignment to center, and padding to 10 pixels.
- Lines 61 through 70 perform all the necessary steps to add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 77 through 91. It is an inner class named `CalcButtonHandler`. The `handle` method appears in lines 80 through 90. The statement in line 83 gets the value that the user entered into the `kiloTextField` control, converts it to a `Double`, and assigns it to the `kilometers` variable. Line 86 converts the kilometers to miles and assigns the result to the `miles` variable. Line 89 displays the results in the `resultLabel` control.



## Checkpoint

1. 11.33 In what package is the `TextField` class?
2. 11.34 How do you retrieve the text that the user has entered into a `TextField` control?

# 11.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events

## Concept:

Event handler code can sometimes be simplified with anonymous inner classes or lambda expressions.

## Using Anonymous Inner Classes to Create Event Handlers

If you look back at [Code Listing 11-11](#) and [Code Listing 11-12](#), you will notice that each program's event handler classes are instantiated only once: when the event handler is registered with a control. Recall from [Chapter 9](#) that when a class is instantiated only once in a program, you can use an anonymous inner class instead of a regular class definition. You use the new operator to simultaneously define an anonymous inner class, and create an instance of it. [Code Listing 11-13](#) shows an example of how we can modify the Kilometer Converter application to use an anonymous inner class as the calcButton control's event handler.



**VideoNote** Using Anonymous Inner Classes as Event Handlers

## Code Listing 11-13

# (AnonInnerClassKiloConverter.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextField;
10 import javafx.scene.control.Button;
11 import javafx.event.EventHandler;
12 import javafx.event.ActionEvent;
13
14 /**
15 * Kilometer Converter application that uses an
16 * anonymous inner class to create an event handler.
17 */
18
19 public class AnonInnerClassKiloConverter extends Application
20 {
21 // Fields
22 private TextField kiloTextField;
23 private Label resultLabel;
24
25 public static void main(String[] args)
26 {
27 // Launch the application.
28 launch(args);
29 }
30
31 @Override
32 public void start(Stage primaryStage)
33 {
34 // Create a Label to display a prompt.
35 Label promptLabel = new Label("Enter a distance in kilometers");
36
37 // Create a TextField for input.
38 kiloTextField = new TextField();
39
40 // Create a Button to perform the conversion.
41 Button calcButton = new Button("Convert");
42
43 // Create an event handler.
44 calcButton.setOnAction(new EventHandler<ActionEvent>()
```

```

45 {
46 @Override
47 public void handle(ActionEvent event)
48 {
49 // Get the kilometers.
50 Double kilometers = Double.parseDouble(kiloTextField.ge
51
52 // Convert the kilometers to miles.
53 Double miles = kilometers * 0.6214;
54
55 // Display the results.
56 resultLabel.setText(String.format("%,.2f miles", miles)
57 }
58 });
59
60 // Create an empty Label to display the result.
61 resultLabel = new Label();
62
63 // Put the promptLabel and the kiloTextField in an HBox.
64 HBox hbox = new HBox(10, promptLabel, kiloTextField);
65
66 // Put the HBox, calcButton, and resultLabel in a VBox.
67 VBox vbox = new VBox(10, hbox, calcButton, resultLabel);
68
69 // Set the VBox's alignment to center.
70 vbox.setAlignment(Pos.CENTER);
71
72 // Set the VBox's padding to 10 pixels.
73 vbox.setPadding(new Insets(10));
74
75 // Create a Scene.
76 Scene scene = new Scene(vbox);
77
78 // Add the Scene to the Stage.
79 primaryStage.setScene(scene);
80
81 // Set the stage title.
82 primaryStage.setTitle("Kilometer Converter");
83
84 // Show the window.
85 primaryStage.show();
86 }
87 }
```

In this version of the program, we have eliminated the definition of the CalcButtonHandler class. Instead, in lines 44 through 58, we instantiate an anonymous inner class that implements the EventHandler interface, and we

register that object as the calcButton control's event handler.

# Using Lambda Expressions to Create Event Handlers

Recall our discussion of functional interfaces and lambda expressions in [Chapter 9](#). A functional interface is an interface that has one, and only one, abstract method. The `EventHandler` interface has only one abstract method (the `handle` method), so it is a functional interface. Any time you are writing Java code to instantiate an anonymous class that implements a functional interface, you should consider using a lambda expression instead. A lambda expression is more concise than the code for instantiating an anonymous class. [Code Listing 11-14](#) shows how we can use a lambda expression to further simplify the Kilometer Converter application. The lambda expression appears in lines 43 through 53. If you compare [Code Listing 11-14](#) with the program shown in [Code Listing 11-13](#), you can see that the lambda expression is more concise than the anonymous inner class declaration.



**VideoNote** Using Lambda Expressions as Event Handlers

## Code Listing 11-14 (`LambdaKiloConverter.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextField;
10 import javafx.scene.control.Button;
```

```
11 import javafx.event.EventHandler;
12 import javafx.event.ActionEvent;
13
14 /**
15 * Lambda version of the Kilometer Converter application
16 */
17
18 public class LambdaKiloConverter extends Application
19 {
20 // Fields
21 private TextField kiloTextField;
22 private Label resultLabel;
23
24 public static void main(String[] args)
25 {
26 // Launch the application.
27 launch(args);
28 }
29
30 @Override
31 public void start(Stage primaryStage)
32 {
33 // Create a Label to display a prompt.
34 Label promptLabel = new Label("Enter a distance in kilometer");
35
36 // Create a TextField for input.
37 kiloTextField = new TextField();
38
39 // Create a Button to perform the conversion.
40 Button calcButton = new Button("Convert");
41
42 // Create an event handler.
43 calcButton.setOnAction(event ->
44 {
45 // Get the kilometers.
46 double kilometers = Double.parseDouble(kiloTextField.getText());
47
48 // Convert the kilometers to miles.
49 double miles = kilometers * 0.6214;
50
51 // Display the results.
52 resultLabel.setText(String.format("%.2f miles", miles));
53 });
54
55 // Create an empty Label to display the result.
56 resultLabel = new Label();
57 }
}
```

```

58 // Put the promptLabel and the kiloTextField in an HBox.
59 HBox hbox = new HBox(10, promptLabel, kiloTextField);
60
61 // Put the HBox, calcButton, and resultLabel in a VBox.
62 VBox vbox = new VBox(10, hbox, calcButton, resultLabel);
63
64 // Set the VBox's alignment to center.
65 vbox.setAlignment(Pos.CENTER);
66
67 // Set the VBox's padding to 10 pixels.
68 vbox.setPadding(new Insets(10));
69
70 // Create a Scene.
71 Scene scene = new Scene(vbox);
72
73 // Add the Scene to the Stage.
74 primaryStage.setScene(scene);
75
76 // Set the stage title.
77 primaryStage.setTitle("Kilometer Converter");
78
79 // Show the window.
80 primaryStage.show();
81 }
82 }
```

Although lambda expressions simplify the use of anonymous inner classes, they have limitations. Recall from [Chapter 9](#) that a lambda expression can access only variables that are effectively `final`. Also, lambda expressions can become difficult to read if they contain a lot of code.

There are many different techniques for creating event handlers. In this chapter, you have learned three techniques:

- Writing the definition of an inner class that implements the `EventHandler` interface. Then, instantiating that class, and registering it with a control. You saw this technique in [Code Listings 11-11](#) and [11-12](#).
- Instantiating an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control. You saw this technique in [Code Listing 11-13](#).

- Using a lambda expression to instantiate an anonymous inner class that implements the `EventHandler` interface, and registering the object with a control. You saw this technique in [Code Listing 11-11](#).

Each technique has its own advantages. For example, technique #1, defining an inner class, is advantageous for writing event handlers with a lot of code. Technique #2, instantiating an anonymous inner class, is advantageous for writing event handlers that have only a small amount of code. Technique #3, using a lambda expression, is the best approach for writing event handlers that have a small amount of code, as long as the code accesses variables that are effectively `final`.



## Checkpoint

1. 11.35 Describe three different techniques discussed in this chapter for writing event handlers.

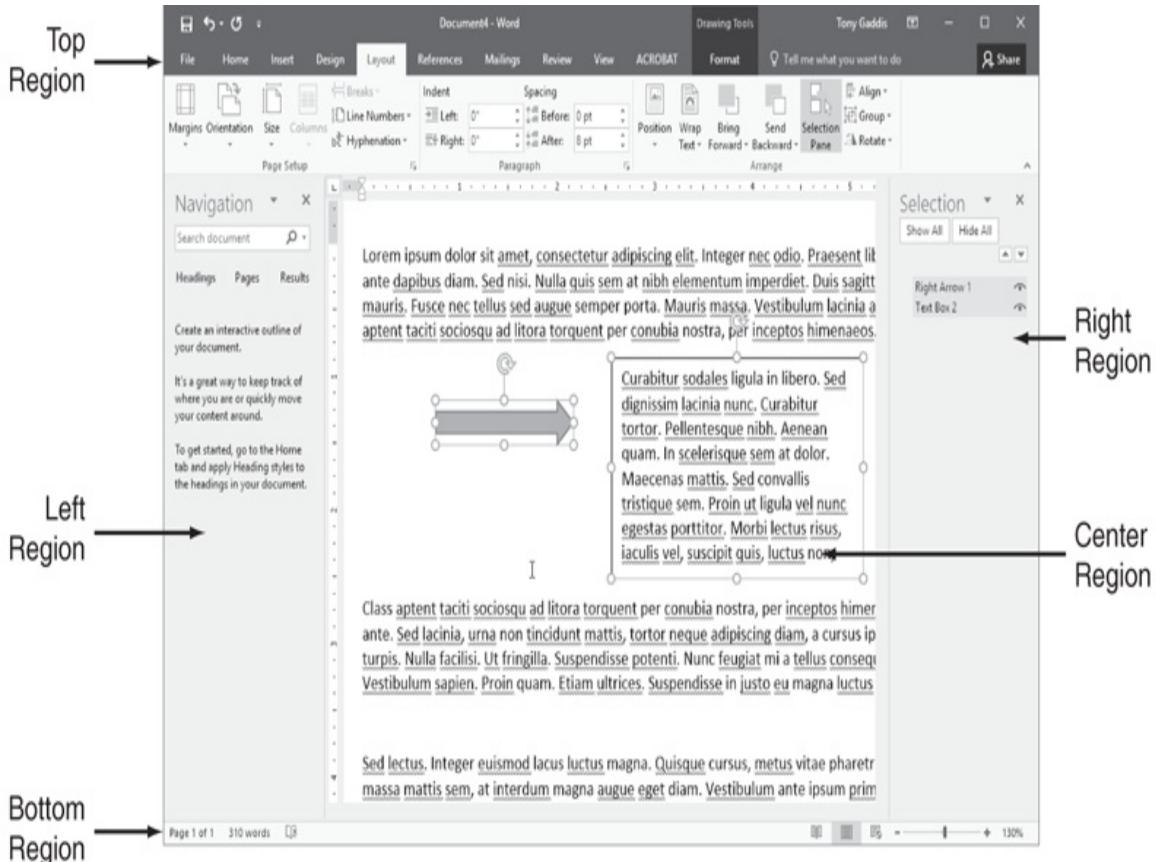
# **11.9 The BorderPane Layout Container**

## **Concept:**

The BorderPane container displays its content in five regions. The regions are known as top, bottom, left, right, and center.

Desktop applications commonly display a GUI with its controls organized in regions. For example, a GUI might display its main content in the center region of the window, while also displaying controls along the window's top, bottom, left, and right regions. [Figure 11-35](#) shows an example using Microsoft Word. The document is displayed in the center of the window, while horizontal menu bars appear along the top, a status bar appears along the bottom, and vertically-arranged elements appear along the left and right edges of the window.

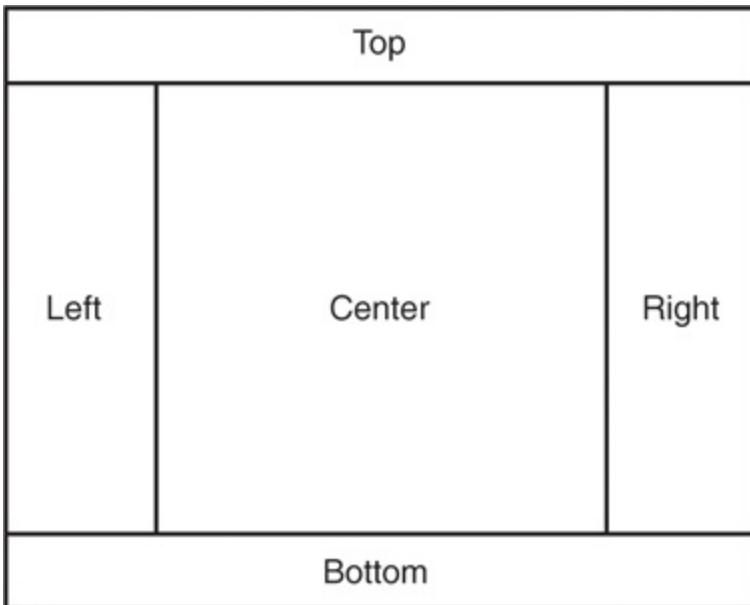
## **Figure 11-35 Typical GUI organization**



[Figure 11-35 Full Alternative Text](#)

JavaFX provides a layout container named `BorderPane` that makes it easy to achieve this style of organization. The `BorderPane` layout container is divided into five regions: top, bottom, left, right, and center. The arrangement of these regions is shown in [Figure 11-36](#). When a node is placed into a `BorderPane` layout container, it must be placed into one of these five regions.

## Figure 11-36 The regions of a BorderPane layout container



Only one object at a time may be placed into a BorderPane region. For this reason, you do not usually put controls directly into a BorderPane region. Typically, you put controls into another type of layout container (such as an HBox or a VBox), then you put that container into one of the BorderPane regions.



## Note:

You don't have to use all of a BorderPane's regions. If a BorderPane region doesn't contain anything, that region simply doesn't appear in the GUI.

To create a BorderPane, you import the `BorderPane` class from the `javafx.scene.layout` package. Then, you create an instance of the `BorderPane` class using one of three constructors described in [Table 11-3](#).

## Table 11-3 BorderPane constructors

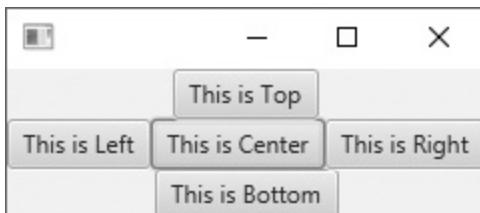
### Constructor

### Description

|                                                                  |                                                                                                                                                                    |
|------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>BorderPane()</code>                                        | The no-arg constructor creates an empty BorderPane container.                                                                                                      |
| <code>BorderPane(<i>center</i>)</code>                           | This constructor accepts one argument. The <code>BorderPane(<i>center</i>)</code> node that is passed as the argument is placed in the BorderPane's center region. |
| <code>BorderPane(<i>center, top, right, bottom, left</i>)</code> | This constructor accepts five nodes as arguments: one to place in each region.                                                                                     |

If you use the first constructor (the no-arg constructor), you will have to use the following BorderPane methods to put nodes in the container's regions: `setCenter`, `setTop`, `setBottom`, `setLeft`, and `setRight`. [Code Listing 11-18](#) demonstrates these methods. The program's output is shown in [Figure 11-37](#).

## Figure 11-37 The BorderPaneDemo1 GUI



[Figure 11-37 Full Alternative Text](#)

## Code Listing 11-18 (BorderPaneDemo1.java)

```

1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Button;
5 import javafx.scene.layout.HBox;
6 import javafx.scene.layout.VBox;
7 import javafx.scene.layout.BorderPane;
```

```
8 import javafx.geometry.Pos;
9
10 /**
11 * A BorderPane Demo
12 */
13
14 public class BorderPaneDemo extends Application
15 {
16 public static void main(String[] args)
17 {
18 // Launch the application.
19 launch(args);
20 }
21
22 @Override
23 public void start(Stage primaryStage)
24 {
25 // Create some buttons.
26 Button centerButton = new Button("This is Center");
27 Button topButton = new Button("This is Top");
28 Button bottomButton = new Button("This is Bottom");
29 Button leftButton = new Button("This is Left");
30 Button rightButton = new Button("This is Right");
31
32 // Add each button to its own layout container.
33 HBox centerHBox = new HBox(centerButton);
34 HBox topHBox = new HBox(topButton);
35 HBox bottomHBox = new HBox(bottomButton);
36 VBox leftVBox = new VBox(leftButton);
37 VBox rightVBox = new VBox(rightButton);
38
39 // Set the alignment for the top and bottom.
40 topHBox.setAlignment(Pos.CENTER);
41 bottomHBox.setAlignment(Pos.CENTER);
42
43 // Create a BorderPane.
44 BorderPane borderPane = new BorderPane();
45
46 // Add the buttons to the BorderPane's regions.
47 borderPane.setCenter(centerHBox);
48 borderPane.setTop(topHBox);
49 borderPane.setBottom(bottomHBox);
50 borderPane.setLeft(leftVBox);
51 borderPane.setRight(rightVBox);
52
53 // Create a Scene with the BorderPane as its root node.
54 Scene scene = new Scene(borderPane);
```

```

55
56 // Add the Scene to the Stage.
57 primaryStage.setScene(scene);
58
59 // Show the window.
60 primaryStage.show();
61 }
62 }
```

Let's take a closer look at the code:

- Lines 26 through 30 create five button controls.
- Lines 33 through 37 put those button controls into their own layout containers. The `centerButton`, `topButton`, and `bottomButton` controls are added to `HBox` containers, and the `leftButton` and `rightButton` controls are added to `VBox` containers.
- Line 44 creates a `BorderPane` container.
- Lines 47 through 51 add the `HBox` and `VBox` containers to the regions of the `BorderPane`.

The second `BorderPane` constructor shown in [Table 11-3](#) accepts one argument: the node to place in the center region. To place nodes in the other regions, you will use the following `BorderPane` methods: `setTop`, `setBottom`, `setLeft`, and `setRight`. The following code snippet shows lines 43 through 50 of the program `BorderPaneDemo2.java`. The program's output is the same as that shown in [Figure 11-37](#).

```

43 // Create a BorderPane with a node in the center.
44 BorderPane borderPane = new BorderPane(centerHBox);
45
46 // Add the buttons to the BorderPane's regions.
47 borderPane.setTop(topHBox);
48 borderPane.setBottom(bottomHBox);
49 borderPane.setLeft(leftVBox);
50 borderPane.setRight(rightVBox);
```

The third constructor shown in [Table 11-3](#) accepts five nodes as arguments: one to place in each region. This is the constructor you will normally want to use when you are going to place nodes in every region, and those nodes exist

when the `BorderPane` is created. The following code snippet shows lines 43 through 46 of the program `BorderPaneDemo3.java`. The program's output is the same as that shown in [Figure 11-37](#).

```
43 // Create a BorderPane.
44 BorderPane borderPane =
45 new BorderPane(centerHBox, topHBox, rightVBox,
46 bottomHBox, leftVBox);
```



## Checkpoint

1. 11.36 What are the names of the regions in a `BorderPane` container?
2. 11.37 Describe the three `BorderPane` constructors.

# 11.10 The ObservableList Interface

## Concept:

Many of the containers in the JavaFX library implement the `ObservableList` interface. It is helpful to learn a few of the methods specified in the interface.

As you spend more time developing JavaFX applications, you will frequently encounter the `ObservableList` interface. An object that implements the `ObservableList` interface is a special type of list that can fire an event handler any time an item in the list changes. Although many of the details of the `ObservableList` interface are beyond the scope of this book, you will commonly work with JavaFX containers that store nodes in an `ObservableList`. Learning a few basic `ObservableList` operations give you more control over the JavaFX containers with which you will be working.

[Table 11-4](#) lists a few of the `observableList` methods. Notice some of the methods are inherited from the `Collection` interface, of which is a `ObservableList` subinterface.

## Table 11-4 A few of the ObservableList methods

| Method                              | Description                                                                                            |
|-------------------------------------|--------------------------------------------------------------------------------------------------------|
| <code>add(<i>item</i>)</code>       | Adds a single item to the list. (This method is inherited from the <code>Collection</code> interface.) |
| <code>addAll(<i>item...</i>)</code> | Adds one or more items to the list, specified by the variable argument list.                           |
| <code>clear()</code>                | Removes all of the items from the list.                                                                |

|                                        |                                                                                                                      |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <code>remove(<i>item</i>)</code>       | Removes the object specified by <i>item</i> from the list. (This method is inherited from the Collection interface.) |
| <code>removeAll(<i>item</i>...)</code> | Removes one or more items to the list, specified by the variable argument list.                                      |
| <code>setAll(<i>item</i>...)</code>    | Clears the current contents of the list and adds all of the items specified by the variable argument list.           |
| <code>size()</code>                    | Returns the number of items in the list. (This method is inherited from the Collection interface.)                   |

For example, layout containers keep their children nodes in an `ObservableList`. All layout containers have a method named `getChildren()` that returns their `ObservableList` of nodes. The following code snippet shows how you can create an empty `HBox` container, then use the `ObservableList`'s `addAll` method to add nodes to the `HBox`:

```
// Create three Label controls.
Label label1 = new Label("This is label1.");
Label label2 = new Label("This is label2.");
Label label3 = new Label("This is label3.");
// Create an empty HBox container.
HBox hbox = new HBox();
// Add the Label controls to the HBox.
hbox.getChildren().addAll(label1, label2, label3);
```

The following code shows how to remove `label1` from the `HBox`:

```
hbox.getChildren().remove(label1);
```

## 11.11 Common Errors to Avoid

- Forgetting to extend the `Application` class in your JavaFX application. The `Application` class in the `javafx.application` package must be extended by all JavaFX applications.
- Forgetting to specify the `file:` protocol when loading an image file with the `Image` class. When creating an instance of the `Image` class and passing a file name to the constructor, be sure to include the `file:` protocol along with the file name.
- Trying to display an image using only the `Image` class. The `Image` class loads an image file, but you also need the `ImageView` class to add the image to your scene graph.
- Resizing an image but forgetting to preserve the aspect ratio. If you resize an image without preserving its aspect ratio, the image can appear vertically or horizontally stretched. The `ImageView` class has a method named `setPreserveRatio` you can call to make sure that an image's aspect ratio is preserved.
- Forgetting to register an event handler. Even if you write an event handler class, it will not execute unless it has been registered with the correct control.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. The primary purpose of this type of control is to display text.
  1. Button
  2. Label
  3. Message
  4. Text
  
2. This type of control appears as a rectangular region that can accept keyboard input from the user.
  1. Button
  2. Label
  3. TextField
  4. InputField
  
3. Typically, when the user clicks this type of control, an action takes place.
  1. Button
  2. Label
  3. TextField
  4. Switch

4. In memory, the GUI objects in a scene are organized as nodes in this type of structure.
  1. scene array
  2. scene graph
  3. scene tree
  4. scene stack
5. There can be only one of these in a scene. It is the parent of all the other nodes.
  1. leaf node
  2. master node
  3. root node
  4. branch node
6. This type of node can have other nodes as children.
  1. leaf node
  2. master node
  3. anchor node
  4. branch node
7. This type of node cannot have other nodes as children.
  1. leaf node
  2. root node
  3. container node

4. branch node
8. All JavaFX applications must extend the class.
  1. JavaFX
  2. Application
  3. GUI
  4. Window
9. This container arranges its contents in a single, vertical row.
  1. VerticalBox
  2. VBox
  3. GridPane
  4. HBox
10. You use this class to load an image file into memory.
  1. ImageView
  2. ImageLoader
  3. Image
  4. Img
11. You use this class to actually display an image.
  1. ImageView
  2. ImageLoader
  3. Image

4. Img

12. The EventHandler interface specifies a method named

- 1. register
- 2. onClick
- 3. fire
- 4. handle

13. The BorderPane layout container has five regions known as which of the following?

- 1. north, south, east, west, center
- 2. top, bottom, left, right, center
- 3. up, down, left, right, middle
- 4. I, II, III, IV, and V

14. What type of object does a layout container's getChildren() method return?

- 1. ExpandableList
- 2. MoveableList
- 3. ArrayList
- 4. ObservableList

15. True or False: The Stage class is in the javafx.scene package.

16. True or False: The HBox container makes all of the nodes it contains of the same height.

17. True or False: You can nest a layout container inside of another layout container.
18. True or False: Only one object at a time may be placed into a BorderPane region.

## Find the Error

Find the error in each of the following code segments:

1. // This code has an error!

```
@Override
public void start(Scene primaryScene)
{
 primaryScene.show();
}
```
2. // This code has an error!

```
Label messageLabel = new Label("Hello World");
HBox hbox = new HBox();
hbox.add(messageLabel);
```
3. Assume hbox is an HBox container:

```
// This code has an error!
hbox.setAlignment(CENTER);
```
4. // This code has an error!

```
ImageView view = new ImageView("file:HotAirBalloon.jp
```
5. // This code has an error!

```
myImageView.setWidth(100);
myImageView.setHeight(100);
```

## Algorithm Workbench

1. Write a statement, after the following code, that creates an HBox container, and adds the label1, label2, and label3 controls to it:

```
Label label1 = new Label("One");
Label label2 = new Label("Two");
Label label3 = new Label("Three");
```

2. Assume the `hbox` variable references an `HBox` container. Write a statement that creates a `Scene` object, adds `hbox` as the root node, and sets the size of the scene to 300 pixels wide by 200 pixels high.
3. Assume `primaryStage` references the `Stage` object, and `scene` references your `Scene` object. Write a statement that adds the `scene` to the stage.
4. Assume the `hbox` variable references an `HBox` container. Write a statement that center-aligns all controls in the `hbox` container.
5. Assume your JavaFX application is in the same directory as an image file named `Cat.png`. Write the code to create the `Image` and `ImageView` objects you will need to display the image.
6. Write a statement that creates an `Image` object, loading the `Cat.png` file from the following Internet location: <http://www.greendale.edu/images/>
7. Write a statement, after the following code, that creates an `HBox` container, adds the `label1`, `label2`, and `label3` controls to it, and has 10 pixels of spacing between the controls.

```
Label label1 = new Label("One");
Label label2 = new Label("Two");
Label label3 = new Label("Three");
```

8. Assume the `vbox` variable references a `VBox` container. Write a statement that adds 10 pixels of padding around the inside edge of the container.
9. Assume `gridPane` is the name of a `GridPane` container, and `button` is the name of a `Button` control. Write a statement that adds `button` to `gridPane` at column 2, row 5.
10. Write a statement that creates a `Button` control with the text `Click Me` displayed on the button.

11. Assume a JavaFX application has a Button control named `myButton`, and a Label control named `outputLabel`. Write an event handler class that can be used with the Button control. The event handler class should display the string “Hello World” in `outputLabel`.
12. In question 11, you wrote the code for an event handler class. Write a statement that registers an instance of the class with the `myButton` control.
13. In question 11, you wrote the code for an event handler class, and in question 12, you wrote the code to register an instance of that class with the `myButton` control. Rewrite the code as a lambda expression.
14. Assume `borderPane` is the name of an existing BorderPane object. Write statements that add the following existing components to the BorderPane:
  - Add `label1` to the top region
  - Add `label2` to the bottom region
  - Add `label3` to the center region
  - Add `label4` to the left region
  - Add `label5` to the right region

## Short Answer

1. What is an event-driven program?
2. What is the purpose of the `Application` class’s `launch` method?
3. What is the purpose of the `Application` class’s abstract `start` method?
4. What purpose do layout containers serve?

5. What package are the `Label`, `Button`, and `TextField` classes in?
6. What two classes do you use to display an image?
7. What is an image's aspect ratio? How do you make sure it is preserved when you resize an image?
8. Which layout container arranges controls in a horizontal row? Which layout container arranges controls in a vertical row? Which layout container arranges controls in rows and columns?
9. How do you retrieve the text that the user has entered into a `TextField` control?
10. Describe three different techniques discussed in this chapter for writing event handlers.
11. Which layout container divides a container into regions known as top, bottom, left, center, and right?

# Programming Challenges

## 1. Latin Translator

Look at the following list of Latin words and their meanings:



**VideoNote** The Latin Translator Problem

| Latin    | English |
|----------|---------|
| sinister | left    |
| dexter   | right   |
| medium   | center  |

Create a JavaFX application that translates the Latin words to English. The form should have three Buttons, one for each Latin word. When the user clicks a Button, the application should display the English translation in a Label.

## 2. Name Formatter

Create a JavaFX application that lets the user enter the following pieces of data:

- The user’s first name
- The user’s middle name
- The user’s last name
- The user’s preferred title (Mr., Mrs., Ms., Dr., and so on.)

Assume the user has entered the following data:

- First name: *Kelly*

- Middle name: *Jane*
- Last name: *Smith*
- Title: *Ms.*

The application should have buttons that display the user's name formatted in the following ways:

- *Ms. Kelly Jane Smith*
- *Kelly Jane Smith*
- *Kelly Smith*
- *Smith, Kelly Jane, Ms.*
- *Smith, Kelly Jane*
- *Smith, Kelly*

### 3. Tip, Tax, and Total

Create a JavaFX application that lets the user enter the food charge for a meal at a restaurant. When a button is clicked, the application should calculate and display the amount of an 18 percent tip on the total food charge, 7 percent sales tax, and the total of all three amounts.

### 4. Property Tax

A county collects property taxes on the assessment value of property, which is 60 percent of the property's actual value. If an acre of land is valued at \$10,000, its assessment value is \$6,000. The property tax is then \$0.64 for each \$100 of the assessment value. The tax for the acre assessed at \$6,000 will be \$38.40. Create a GUI application that displays the assessment value and property tax when a user enters the actual value of a property.

### 5. Heads or Tails

Create a JavaFX application that simulates a coin being tossed. When the user clicks a button, the application should generate a random number in the range of 0 to 1. If the number is 0, the coin has landed on “heads,” and if the number is 1, the coin has landed on “tails.” Use an `ImageView` control, and the coin images in this book’s Student Sample Programs to display the side of the coin when it is tossed.

## 6. Dice Simulator

Create a JavaFX application that simulates rolling a pair of dice. When the user clicks a button, the application should generate two random numbers, each in the range of 1 through 6, to represent the value of the dice. Use `ImageView` controls to display the dice. (In the Student Sample Programs, you will find six images named `Die1.png`, `Die2.png`, `Die3.png`, `Die4.png`, `Die5.png`, and `Die6.png` that you can use in the `ImageView` controls.)

## 7. Travel Expenses

Create a GUI application that calculates and displays the total travel expenses of a businessperson on a trip. Here is the information that the user must provide:

- Number of days on the trip
- Amount of airfare, if any
- Amount of car rental fees, if any
- Number of miles driven, if a private vehicle was used
- Amount of parking fees, if any
- Amount of taxi charges, if any
- Conference or seminar registration fees, if any
- Lodging charges, per night

The company reimburses travel expenses according to the following policy:

- \$47 per day for meals
- Parking fees up to \$20.00 per day
- Taxi charges up to \$40.00 per day
- Lodging charges up to \$195.00 per day
- If a private vehicle is used, \$0.42 per mile driven

The application should calculate and display the following:

- Total expenses incurred by the businessperson
- The total allowable expenses for the trip
- The excess that must be paid by the businessperson, if any
- The amount saved by the businessperson if the expenses were under the total allowed

## 8. Joe's Automotive

Joe's Automotive performs the following routine maintenance services:

- Oil change: \$35.00
- Lube job: \$25.00
- Radiator flush: \$50.00
- Transmission flush: \$120.00
- Inspection: \$35.00
- Muffler replacement: \$200.00

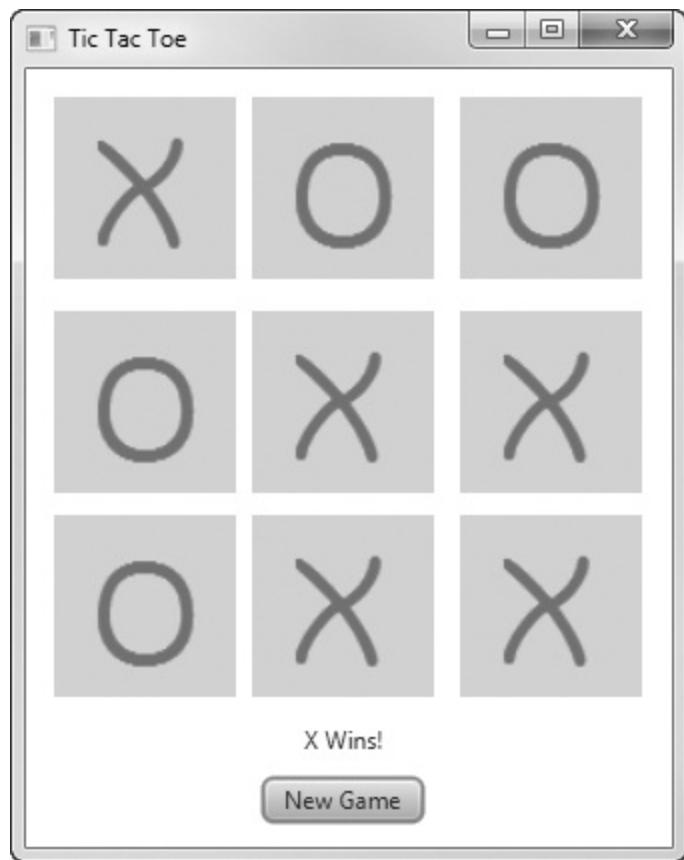
- Tire rotation: \$20.00

Joe also performs other nonroutine services and charges for parts and for labor (\$60 per hour). Create a GUI application that displays the total for a customer's visit to Joe's.

## 9. Tic-Tac-Toe Simulator

Create a JavaFX application that simulates a game of tic-tac-toe. [Figure 11-38](#) shows an example of the application's form. The form shown in the figure uses eight `ImageView` controls to display the Xs and Os. (You will find images for the X and the O in the book's Student Sample Files.)

# Figure 11-38 The tic-tac-toe application



[Figure 11-38 Full Alternative Text](#)

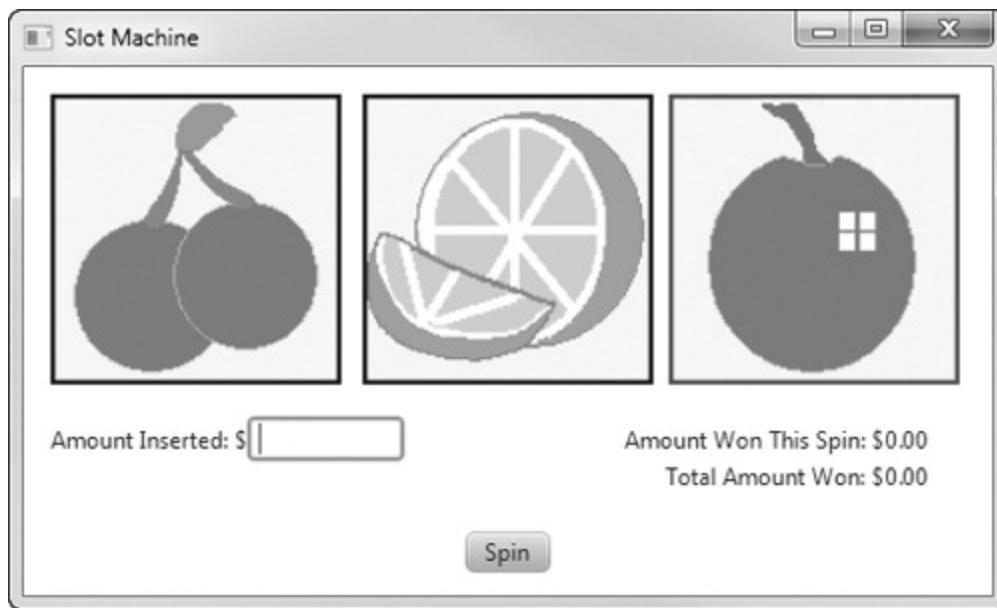
The application should use a two-dimensional `int` array to simulate the game board in memory. When the user clicks the *New Game* button, the application should step through the array, storing a random number in the range of 0 through 1 in each element. The number 0 represents the letter O, and the number 1 represents the letter X. The application's window should then be updated to display the game board. The application should display a message indicating whether player X won, player Y won, or the game was a tie.

## 10. Slot Machine Simulation

A slot machine is a gambling device into which the user inserts money then pulls a lever (or presses a button). The slot machine then displays a set of random images. If two or more of the images match, the user wins an amount of money that the slot machine dispenses back to the user.

Create a JavaFX application that simulates a slot machine. [Figure 11-39](#) shows an example of how the GUI should look. The application should let the user enter into a `TextField` the amount of money he or she is inserting into the machine. When the user clicks the *Spin* button, the application should display three randomly selected symbols. (Slot machines traditionally display fruit symbols. You will find a set of fruit symbol images in the Student Sample Programs.) The program should also display the amount that the user won for the spin, and the total amount won for all spins.

## Figure 11-39 Slot machine application



[Figure 11-39 Full Alternative Text](#)

The amount won for a spin is determined in the following way:

- If none of the randomly displayed images match, the user has won \$0.
- If two of the images match, the user has won two times the amount

entered.

- If three of the images match, the user has won three times the amount entered.

# Chapter 12 JavaFX: Advanced Controls

## Topics

1. [12.1 Styling JavaFX Applications with CSS](#)
2. [12.2 RadioButton Controls](#)
3. [12.3 CheckBox Controls](#)
4. [12.4 ListView Controls](#)
5. [12.5 ComboBox Controls](#)
6. [12.6 Slider Controls](#)
7. [12.7 TextArea Controls](#)
8. [12.8 Menus](#)
9. [12.9 The FileChooser Class](#)
10. [12.10 Using Console Output to Debug a GUI Application](#)
11. [12.11 Common Errors to Avoid](#)

# 12.1 Styling JavaFX Applications with CSS

## Concept:

You can use Cascading Style Sheet rules to specify how a JavaFX application should appear.

CSS, which stands for *Cascading Style Sheets*, is a simple language that specifies how a user interface should appear. CSS was originally created for Web designers. It gives designers a way to separate a webpage's content from its visual design. In other words, a webpage can use HTML to specify *what* information to display, and CSS to specify *how* that information should appear.



### VideoNote JavaFX and CSS

JavaFX supports CSS. When developing a JavaFX application, you can use CSS to specify many of the user interface's visual properties. For example, you can use CSS to specify font characteristics, background colors, background images, text alignment, border thickness, and much more.

When using CSS, you typically create one or more style sheets to accompany your JavaFX application. A *style sheet* is simply a text file that contains one or more *style definitions*, written in the following general format:

```
selector {
 property: value;
 property: value;
}
```

In the general format, *selector* is a name that determines the node or nodes that are affected by the style definition. Inside the braces, one or more *style rules* appear. A style rule takes the form

```
property: value;
```

where *property* is the name of a property, and *value* is a value to assign to that property. Notice a colon separates the property name from the value, and the style rule ends with a semicolon. Here is an example of style definition:

```
.label {
 -fx-font-size: 20pt;
}
```

In this style definition, `.label` is the selector. Selector names that begin with a period are called *type selectors*, and they correspond to specific types of nodes in the JavaFX application. For example, the `.label` selector corresponds to `Label` controls. As a result, this style definition will be applied to all `Label` controls in the application. Inside the braces, we have one style rule that sets the `-fx-font-size` property to 20 points. So, this style definition specifies that all `Label` controls should display text in a 20-point font.



## Note:

In the previous example, notice the property name begins with `-fx-`. In JavaFX, all CSS properties begin with `-fx-`.

# Type Selector Names

You just learned that selector names beginning with a period are type selectors, and they correspond to specific types of JavaFX nodes. The type selector names are very similar to the JavaFX class names to which they correspond. [Table 12-1](#) shows some examples. (Many of the classes shown in the table will be discussed later in this chapter.)

# Table 12-1 Some of the JavaFX nodes and their corresponding CSS type selectors

| Class Name in JavaFX | Corresponding Type Selector |
|----------------------|-----------------------------|
| Button               | .button                     |
| CheckBox             | .check-box                  |
| CheckMenuItem        | .check-menu-item            |
| ComboBox             | .combo-box                  |
| ImageView            | .image-view                 |
| Label                | .label                      |
| ListView             | .list-view                  |
| Menu                 | .menu                       |
| MenuBar              | .menu-bar                   |
| MenuItem             | .menu-item                  |
| RadioButton          | .radio-button               |
| RadioMenuItem        | .radio-menu-item            |
| Slider               | .slider                     |
| TextArea             | .text-area                  |
| TextField            | .text-field                 |

As you can see from the table, there are some differences between the type selector names and the JavaFX class names. As previously mentioned, the type selectors begin with a period. After the period, the type selector names are written in all lowercase letters. When the JavaFX class name is composed of two words, the two words will be separated by a hyphen in the type selector name. So, the type selector for the CheckBox class is .check-box, and the type selector for the TextField class is .text-field.

## Style Properties

Each type of node in JavaFX has a set of style properties you can use. For example, [Table 12-2](#) shows a few of the style properties for the Labeled

class, and [Table 12-3](#) shows a few properties for the Region class. The Label class, the Button class, the CheckBox class, and the RadioButton class all inherit from the Region class and the Labeled class, so the properties in both of these tables may be used with Label, Button, CheckBox, and RadioButton controls. (The CheckBox and RadioButton controls will be discussed later in this chapter.)

## Table 12-2 A few CSS style properties for the Labeled class (inherited by the Label and Button classes)

| Property           | Description                               | Possible Values                                                                                                                                  |
|--------------------|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| -fx-text-alignment | Specifies the control's text alignment    | left<br>center<br>right<br>justify                                                                                                               |
| -fx-font-family    | Specifies the font for the control's text | Can be the name of a font installed on the local system, or one of the following generic fonts:<br><br>serif<br>sans-serif<br>cursive<br>fantasy |

`monospace`

Can be a number specified in points (pt) or pixels (px).

-fx-font - Specifies the size of

size control's text

12pt

18px

Specifies the style normal

-fx-font - of control's text italic

style (normal, italic, or oblique) oblique

You can specify one of these predefined values:

normal

bold

bolder

lighter

-fx-font - Specifies the weight

weight of the control's text

Alternatively, you can specify one of these numeric values: 100, 200, 300, 400, 500, 600, 700, 800, or 900.

When using the numeric values, 400 is the same weight as `normal`, and 700 is the same weight as `bold`.

Can be a hexadecimal color code,

`-fx-text-fill` Specifies the color or a named color (see the discussion of color codes later in this section)

## Table 12-3 A few CSS style properties for the Region class (inherited by the Label and Button classes)

| Property                          | Description                                             | Possible Values                                                                                                                                                       |
|-----------------------------------|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-fx-border-style</code>     | Specifies the type of border to draw around the control | There are many possible settings for this property, but the common ones are:<br><code>none</code><br><code>solid</code><br><code>dotted</code><br><code>dashed</code> |
| <code>-fx-border-width</code>     | Specifies the width of the control's border             | Typically expressed as a number of pixels, such as <code>3px</code>                                                                                                   |
| <code>-fx-border-color</code>     | Specifies the color of the control's border             | Can be a hexadecimal color code, or a named color (see the discussion of color codes later in this section)                                                           |
| <code>-fx-background-color</code> | Specifies the control's                                 | Can be a hexadecimal color code, or a named color (see the discussion of color codes later in                                                                         |

background color this section)

Here is an example style definition that uses several of these properties:

```
.label {
 -fx-font-family: cursive;
 -fx-font-size: 14pt;
 -fx-font-style: italic;
 -fx-font-weight: bold;
 -fx-border-style: dotted;
}
```

If we use this style definition with a JavaFX application, its style rules will be applied to all `Label` controls in the user interface.

## Applying a Stylesheet to a JavaFX Application

As previously mentioned, a stylesheet is simply a text file that contains style definitions. Style sheets are usually saved with the `.css` file extension. To apply a stylesheet to a JavaFX application, you save the stylesheet in the same directory as the JavaFX application. Then, in the application's code, you add the stylesheet to the scene. For example, suppose we have a stylesheet named `mystyles.css`, and in our Java code, the `scene` variable references our `Scene` object. We would use the following statement to add the stylesheet to the scene:

```
scene.getStylesheets().add("mystyles.css");
```

The `Scene` class's `getStylesheets()` method returns an object containing the scene's collection of stylesheets, and that object's `add` method adds the specified stylesheet to the collection.

Let's look at an example. [Code Listing 12-1](#) shows the contents of the `demo1.css` file, and [Code Listing 12-2](#) shows a JavaFX application that applies that stylesheet to its scene. The program's output is shown in [Figure](#)

[12-1.](#)

## Figure 12-1 Output of CSSDemo1.java



## Code Listing 12-1 (demo1.css)

```
1 .label {
2 -fx-text-alignment: center;
3 -fx-font-family: cursive;
4 -fx-font-size: 14pt;
5 -fx-font-style: italic;
6 -fx-font-weight: bold;
7 -fx-border-style: dotted;
8 }
```

## Code Listing 12-2 (CSSDemo1.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.scene.control.Label;
6 import javafx.geometry.Insets;
7 import javafx.geometry.Pos;
8
9 public class CSSDemo1 extends Application
10 {
11 public static void main(String[] args)
12 {
13 launch(args);
14 }
```

```

15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Create a Label control.
20 Label myLabel = new Label("Hello world!");
21
22 // Put the Label in a VBox.
23 VBox vbox = new VBox(myLabel);
24 vbox.setAlignment(Pos.CENTER);
25 vbox.setPadding(new Insets(10));
26
27 // Create a Scene and display it.
28 Scene scene = new Scene(vbox);
29 scene.getStylesheets().add("demo1.css");
30 primaryStage.setScene(scene);
31 primaryStage.show();
32 }
33 }
```

Let's look at another example. The stylesheet shown in [Code Listing 12-3](#) has style definitions for Label and Button controls. The program in [Code Listing 12-4](#) applies the stylesheet, and [Figure 12-2](#) shows the program's output.

## Figure 12-2 Output of CSSDemo2.java



## Code Listing 12-3 (demo2.css)

```

1 .label {
2 -fx-font-family: cursive;
3 -fx-font-size: 14pt;
4 -fx-font-style: italic;
5 -fx-border-style: dashed;
```

```
6 }
7
8 .button {
9 -fx-font-size: 10pt;
10 -fx-font-weight: 900;
11 }
```

## Code Listing 12-4 (CSSDemo2.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.geometry.Insets;
8 import javafx.geometry.Pos;
9
10 public class CSSDemo2 extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Create a Label and some Buttons.
21 Label label = new Label("This is a Label");
22 Button button1 = new Button("Button 1");
23 Button button2 = new Button("Button 2");
24
25 // Put the controls in an HBox.
26 HBox hbox = new HBox(10, label, button1, button2);
27 hbox.setAlignment(Pos.CENTER);
28 hbox.setPadding(new Insets(10));
29
30 // Create a Scene and display it.
31 Scene scene = new Scene(hbox);
32 scene.getStylesheets().add("demo2.css");
33 primaryStage.setScene(scene);
34 primaryStage.show();
35 }
36 }
```

## Tip:

In a JavaFX application, CSS styling is applied at runtime. If you make any changes to the application's stylesheet, you do not have to recompile the Java application. The changes will take effect the next time you run the application.

## Tip:

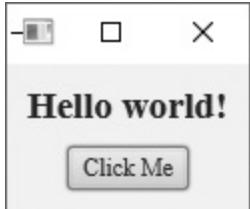
JavaFX supports a large number of CSS properties (more than we can cover in this book). Part of the process of creating stylesheets is determining which properties are available for the nodes in your application. Oracle publishes a comprehensive CSS reference guide which documents all of the available properties. You can find the guide at:

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/>

## Applying Styles to the Root Node

If you want to apply styles to all of the nodes in a scene, use the `.root` selector. [Code Listing 12-5](#) shows an example stylesheet with two style definitions: one for the root node, and another for `Label` controls. The program in [Code Listing 12-6](#) applies the stylesheet, and [Figure 12-3](#) shows the program's output. Notice both the `Label` and the `Button` display their text in a serif font. This is because it is specified by the `.root` style definition.

## Figure 12-3 Output of CSSDemo3.java



## Code Listing 12-5 (demo3.css)

```
1 .root {
2 -fx-font-family: serif;
3 }
4
5 .label {
6 -fx-font-size: 14pt;
7 -fx-font-weight: bold;
8 }
```

## Code Listing 12-6 (CSSDemo3.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.geometry.Insets;
8 import javafx.geometry.Pos;
9
10 public class CSSDemo3 extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Create a Label and a Button.
21 Label myLabel = new Label("Hello world!");
22 Button myButton = new Button("Click Me");
```

```

23
24 // Put the controls in a VBox.
25 VBox vbox = new VBox(10, myLabel, myButton);
26 vbox.setAlignment(Pos.CENTER);
27 vbox.setPadding(new Insets(10));
28
29 // Create a Scene and display it.
30 Scene scene = new Scene(vbox);
31 scene.getStylesheets().add("demo2.css");
32 primaryStage.setScene(scene);
33 primaryStage.show();
34 }
35 }
```

If a style definition for a specific type of node contradicts a style rule in the `.root` definition, the more specific style definition takes precedence over the `.root` definition. To demonstrate, look at the style sheet in [Code Listing 12-7](#). Notice the `.root` definition specifies a font size of 20 points, and the `.label` definition specifies a font size of 10 points. If an application uses this style sheet, its `Label` controls will display text at 10 points, and all other controls will display text at 20 points.

## Code Listing 12-7 (`demo4.css`)

```

1 .root {
2 -fx-font-size: 20pt;
3 }
4
5 .label {
6 -fx-font-size: 10pt;
7 -fx-font-weight: bold;
8 }
```

## Specifying Multiple Selectors in the Same Style Definition

You can list multiple selectors, separated by commas, in the first line of a style definition. [Code Listing 12-8](#) shows an example. Notice line 1 lists the

.label selector, followed by the .button selector. As a result, the style rules will be applied to Label controls and Button controls.

## Code Listing 12-8 (demo5.css)

```
1 .label, .button{
2 -fx-font-family: cursive;
3 -fx-font-size: 14pt;
4 -fx-font-style: italic;
5 }
```

## Working with Colors

CSS uses the *RGB* color system to define colors. In the RGB color system, all colors are created by mixing various shades of red, green, and blue. For example, if you mix bright red, bright green, and no blue you get yellow. If you mix bright red, bright blue, and no green, you get purple. White is created when you combine all three colors at their maximum brightness, and black is created when there is no red, green, or blue.

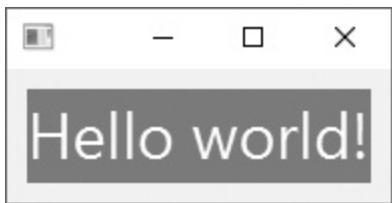
When you define a color in CSS, you specify an integer value in the range of 0 through 255 for each of the color components (red, green, and blue). The higher the number for a color component, the brighter that color component will be. The most common way to specify a color is with a hexadecimal number, prefixed with the # character. The general format of a hexadecimal color value is:

#RRGGBB

where RR is the hexadecimal value for red, GG is the hexadecimal value for green, and BB is the hexadecimal value for blue. Each of these color components must be a hexadecimal number in the range of 00 through FF. For example, the value #05AAFF specifies 05 for red, AA for green, and FF for blue. When these color components are combined, we get a light shade of blue.

To demonstrate, look at the style sheet in [Code Listing 12-9](#). It defines a style for Label controls. The style definition specifies a font size of 20 points, a text fill color of #FFFFFF (white), and a background color of #909090 (gray). [Figure 12-4](#) shows an example of how a Label appears with this style applied.

## Figure 12-4 Example Label with styles applied



## Code Listing 12-9 (`demo6.css`)

```
1 .label {
2 -fx-font-size: 24pt;
3 -fx-text-fill: #FFFFFF;
4 -fx-background-color: #909090;
5 }
```

If you aren't comfortable with hexadecimal values, you can use *functional RGB notation* to specify colors. The general format is:

`rgb(red, green, blue)`

In this notation, the values given for red, green, and blue can either be an integer in the range of 0 through 255, or a percentage in the range of 0 percent through 100 percent. For example, the notation `rgb(255, 0, 127)` specifies 255 for red, 0 for green, and 127 for blue. And, the notation `rgb(50%, 20%, 90%)` specifies that red should have 50 percent intensity, green should have 20 percent intensity, and blue should have 90 percent intensity.

To demonstrate, look at the style sheet in [Code Listing 12-10](#). The `.label` style definition uses functional RGB notation to specify the same color values as previously shown in [Code Listing 12-9](#). The style definition specifies the following:

- A font size of 24 points
- A text fill color where red = 255, green = 255, and blue = 255
- A background where red = 144, green = 144, and blue = 144

## Code Listing 12-10 (`demo7.css`)

```
1 .label {
2 -fx-font-size: 24pt;
3 -fx-text-fill: rgb(255, 255, 255);
4 -fx-background-color: rgb(144, 144, 144);
5 }
```

## Named Colors

CSS provides numerous predefined names that represent colors. We refer to these names as the *named colors*. You can use these named colors just as you would use a hexadecimal color number, or a functional RGB expression. At the time of this writing, JavaFX supports 148 named colors. [Table 12-4](#) lists only a few of them. You can see the entire list in Appendix K, which you can download from the computer science portal at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). You can also see them in the official JavaFX CSS Reference Guide, available at:

<https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/>

## Table 12-4 A few of the named colors

|              |         |            |           |
|--------------|---------|------------|-----------|
| antiquewhite | aqua    | black      | blue      |
| chocolate    | crimson | darkgray   | darkgreen |
| gold         | gray    | green      | lavender  |
| maroon       | olive   | orange     | plum      |
| purple       | red     | sandybrown | tan       |
| teal         | violet  | white      | yellow    |

[Code Listing 12-11](#) shows an example style sheet that uses named colors. The .label style definition specifies white as the text fill color, and blue as the background color.

## Code Listing 12-11 (demo8.css)

```
1 .label {
2 -fx-text-fill: white;
3 -fx-background-color: blue;
4 }
```

# Creating a Custom Style Class Name

In a type selector, the name that appears after the period is known as the *style class*. For example, in the .label selector, the style class is label, and in the .button selector, the style class is button. To achieve more flexibility in applying style definitions, JavaFX allows you to create your own custom style classes.

For example, suppose you want some of the Button controls in an application to appear with a black background and white text, and you want other Button controls to appear with a white background and black text. This is possible if you create two custom style classes: one for the black buttons, and another for the white buttons. [Code Listing 12-12](#) shows an example style sheet in which we have created two custom style classes: one named button-black, and another named button-white.

## Code Listing 12-12 (demo9.css)

```
1 .button-black {
2 -fx-background-color: black;
3 -fx-text-fill: white;
4 }
5
6 .button-white {
7 -fx-background-color: white;
8 -fx-text-fill: black;
9 }
```

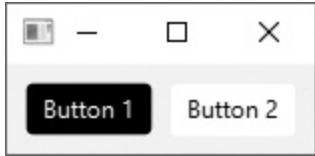
In this style sheet, the `.button-black` style definition specifies a black background with white text. The `.button-white` style definition specifies a white background with black text. To apply either of these style definitions to a node, we must add the desired style class to the appropriate node in our Java application.

For example, suppose we have a `Button` control named `button1`, and we want to apply the `.button-black` style definition to it. We would use the following statement to add the `button-black` style class to `button1`:

```
button1.getStyleClass().add("button-black");
```

Each node keeps an internal collection of style classes. This collection determines which style definitions will be applied to the node. The `getStyleClass()` method returns an object containing the node's collection of style classes, and that object's `add` method adds the specified style class to the collection. (When you pass a style class name as an argument to the `add` method, you do not include the period.) [Code Listing 12-13](#) shows a complete application that uses the custom style classes previously shown. The program's output is shown in [Figure 12-5](#).

## Figure 12-5 Output of CSSDemo9.java



## Code Listing 12-13 (CSSDemo9.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.control.Button;
6 import javafx.geometry.Insets;
7 import javafx.geometry.Pos;
8
9 public class CSSDemo9 extends Application
10 {
11 public static void main(String[] args)
12 {
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Create some Buttons.
20 Button button1 = new Button("Button 1");
21 button1.getStyleClass().add("button-black");
22
23 Button button2 = new Button("Button 2");
24 button2.getStyleClass().add("button-white");
25
26 // Put the Buttons in an HBox.
27 HBox hbox = new HBox(10, button1, button2);
28 hbox.setAlignment(Pos.CENTER);
29 hbox.setPadding(new Insets(10));
30
31 // Create a Scene and display it.
32 Scene scene = new Scene(hbox);
33 scene.getStylesheets().add("demo9.css");
34 primaryStage.setScene(scene);
35 primaryStage.show();
36 }
37 }
```

# ID Selectors

Each node in the scene graph can optionally be assigned a unique identifier known as an *ID*. A node's ID is a string that can be used to identify the node in a style sheet. This is useful when you want to apply a style to one specific node.

You assign an ID to a node by calling the node's `setId` method. Here is an example:

```
Label outputLabel = new Label("File not found!");
outputLabel.setId("label-error");
```

In this example, `outputLabel` is a `Label` control, and we have assigned it the ID "label-error". Next, we create a style definition with the selector `#label-error` in our style sheet. Here is an example:

```
#label-error {
 -fx-background-color: red;
 -fx-text-fill: white;
}
```

Notice the selector in this style definition begins with the # character. This indicates that it is an *ID selector*. The name that follows the # character is assumed to be the ID of a node. As a result, this style definition will be applied only to the node with this specific ID.



## Note:

All node IDs should be unique within a scene graph. Unfortunately, the Java compiler does not check for duplicate IDs, and no exception will be thrown at runtime if two nodes have the same ID. It is your responsibility, as the developer, to make sure that you do not assign the same ID to more than one node.

# Inline Style Rules

Style rules can be applied directly to a node in the application's Java code, using the node's `setStyle` method. You simply pass the style rule, as a string argument, to the method. Here is an example:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt");
```

This code creates a `Label` control named `outputLabel`, and sets the `Label`'s font size to 24 points. Notice the style rule argument does not end with a semicolon. In inline styling, semicolons are required only to separate multiple style rules. Here is an example that passes two style rules to the `setStyle` method:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt; -fx-font-weight: bold")
```

This code creates a `Label` control named `outputLabel`, sets the `Label`'s font size to 24 points, and sets the font's weight to `bold`.

Keep in mind the `setStyle` method removes any styles that were previously applied to the node, and replaces them with the new style rule that is passed as an argument. If you want to add a new style rule to a node's existing set of style rules, call the `getStyleClass().add()` method. The following code snippet shows an example:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt");
outputLabel.getStyleClass().add("-fx-font-weight: bold");
```

This code creates a `Label` control named `outputLabel`, sets the `Label`'s font size to 24 points, and sets the font's weight to `bold`.

Inline styling is convenient when you have only one or two style rules to apply in an application. In most cases, however, it is a better practice to use stylesheets. If you need to change a particular style rule, it will most likely be easier to find that rule in a style sheet, than in the application's Java code.

Also, if you change a style rule that was applied inline, you will have to recompile the Java application. Changing a rule in a style sheet does not require that the application be recompiled.



## Checkpoint

1. 12.1 What is the type selector name that corresponds to each of the following JavaFX types?  
**Button**  
**TextField**  
**Label**  
**ImageView**
2. 12.2 You haven't yet learned about the `MenuButton` control, but you have learned how to convert JavaFX class names to CSS type selectors. What would the type selector name for the `MenuButton` class be?
3. 12.3 Which CSS style property specifies the font for a control's text?
4. 12.4 If you want a control's text to appear in bold, which CSS style property would you set?
5. 12.5 If you want to place a dotted border around a `Label` control, which CSS style property would you set?
6. 12.6 Assume `scene` is the name of a `Scene` object, and `stylesheet.css` is the name of a CSS stylesheet. Write the statement that applies the `stylesheet.css` stylesheet to `scene`.
7. 12.7 If you want to apply styles to all of the nodes in a scene, what selector would you use in a style definition?
8. 12.8 If a style definition for a specific type of node contradicts a style rule in the `.root` definition, which takes precedence—the more specific

style definition, or the .root definition?

9. 12.9 When speaking of colors, what does RGB mean?
10. 12.10 In the hexadecimal color value #0511FF, the FF value specifies which color component?
11. 12.11 In a style definition, what type of selector begins with a # character?

## 12.2 RadioButton Controls

### Concept:

*RadioButton controls normally appear in groups of two or more and allow the user to select one of several possible options.*

RadioButton controls are useful when you want the user to select one choice from several possible options. [Figure 12-6](#) shows a group of RadioButton controls. A RadioButton may be selected or deselected. Each RadioButton has a small circle that appears filled-in when the RadioButton is selected, and appears empty when the RadioButton is deselected.

### Figure 12-6 RadioButton controls

- Choice 1
- Choice 2
- Choice 3



**VideoNote** RadioButton Controls

To create a RadioButton control, you will use the `RadioButton` class, which is in the `javafx.scene.control` package. Once you have written the necessary `import` statement, you will create an instance of the `RadioButton`

class. The following statement shows an example of creating a `RadioButton` object that displays the text *Choice 1*:

```
RadioButton radio1 = new RadioButton("Choice 1");
```

Notice we pass the string we want displayed next to the `RadioButton` as an argument to the `RadioButton` class's constructor. Optionally, you can omit the string argument to create a `RadioButton` control with no text displayed next to it. Here is an example:

```
RadioButton radio1 = new RadioButton();
```

`RadioButton` controls are normally grouped together in a *toggle group*. Only one of the `RadioButton` controls in a toggle group may be selected at any time. Clicking on a `RadioButton` selects it and automatically deselects any other `RadioButton` in the same toggle group. Because only one `RadioButton` in a toggle group can be selected at any given time, the `RadioButton` controls are said to be *mutually exclusive*.

To create a toggle group, you use the `ToggleGroup` class, which is in the `javafx.scene.control` package. Here is an example of a statement that creates an instance of the class:

```
ToggleGroup myToggleGroup = new ToggleGroup();
```

After creating a `ToggleGroup` object, you must call each `RadioButton` control's `setToggleGroup` method to add them to the `ToggleGroup`. The following code snippet shows an example:

```
// Create some RadioButtons.
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
RadioButton radio3 = new RadioButton("Option 3");
// Create a ToggleGroup.
ToggleGroup radioGroup = new ToggleGroup();
// Add the RadioButtons to the ToggleGroup.
radio1.setToggleGroup(radioGroup);
radio2.setToggleGroup(radioGroup);
radio3.setToggleGroup(radioGroup);
```

# Determining in Code Whether a RadioButton Is Selected

To determine whether a RadioButton is selected, you call the RadioButton class's isSelected method. The method returns a boolean value indicating whether the RadioButton is selected. If the RadioButton is selected, the method returns true. Otherwise, it returns false. In the following code, the radio1 variable references a RadioButton. The if statement calls the isSelected method to determine whether the RadioButton is selected.

```
if (radio1.isSelected())
{
 // Code here executes if the radio button is selected. }
```

# Selecting a RadioButton in Code

When you create a group of RadioButton controls, you usually want one of the RadioButtons to be initially selected. You can select a RadioButton in code with the RadioButton class's setSelected method. The setSelected method takes a boolean argument. If you pass true to the method, the RadioButton will be selected just as if the user had clicked it. (If you pass false as an argument to the setSelected method, the RadioButton will be deselected.)

In the following statement, the radio1 variable references a RadioButton. When this statement executes, the RadioButton that is referenced by radio1 will be selected. (Any other RadioButton in the same toggle group that is already selected will be deselected.)

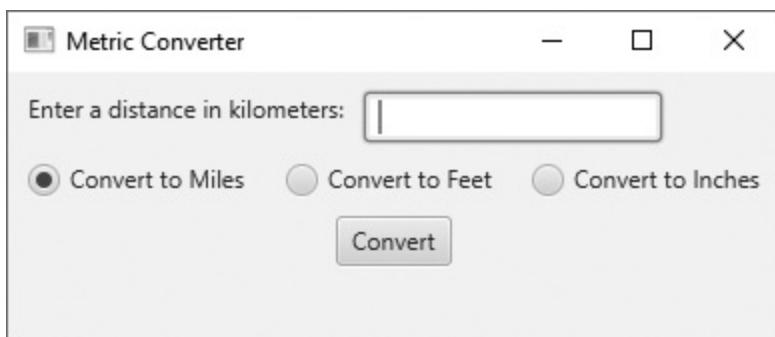
```
radio1.setSelected(true);
```

Let's look at an application that uses RadioButton controls. [Code Listing 12-14](#) shows the code for the Metric Converter application, which converts kilometers to miles, feet, and inches. The conversion formulas are:

*Miles = Kilometers × 0.6214*  
*Feet = Kilometers × 3281.0*  
*Inches = Kilometers × 39370.0*

When the program runs, it displays the window shown in [Figure 12-7](#). The user enters a distance in kilometers, then clicks a RadioButton to select miles, feet, or inches. When the user clicks the Button control, the program performs the selected conversion and displays the result. For example, [Figure 12-8](#) shows three screenshots of the application. In the screens, the user has converted 100 kilometers to miles, feet, and inches.

## Figure 12-7 The Metric Converter application



[Figure 12-7 Full Alternative Text](#)

## Figure 12-8 The Metric Converter application with conversions displayed



[Figure 12-8 Full Alternative Text](#)

## Code Listing 12-14 (**MetricConverter.java**)

```

1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextField;
10 import javafx.scene.control.Button;
11 import javafx.scene.control.RadioButton;
12 import javafx.scene.control.ToggleGroup;
13 import javafx.event.EventHandler;
14 import javafx.event.ActionEvent;
15
16 /**
17 * Metric Converter application
18 */
19

```

```
20 public class MetricConverter extends Application
21 {
22 // Fields
23 private TextField kiloTextField;
24 private Label resultLabel;
25 private RadioButton milesButton;
26 private RadioButton feetButton;
27 private RadioButton inchesButton;
28
29 public static void main(String[] args)
30 {
31 // Launch the application.
32 launch(args);
33 }
34
35 @Override
36 public void start(Stage primaryStage)
37 {
38 // Create a Label to display a prompt.
39 Label promptLabel = new Label("Enter a distance in kilomete
40
41 // Create a TextField for input.
42 kiloTextField = new TextField();
43
44 // Create the RadioButton controls.
45 milesButton = new RadioButton("Convert to Miles");
46 feetButton = new RadioButton("Convert to Feet");
47 inchesButton = new RadioButton("Convert to Inches");
48
49 // Select the milesButton control.
50 milesButton.setSelected(true);
51
52 // Add the RadioButton controls to a ToggleGroup.
53 ToggleGroup radioGroup = new ToggleGroup();
54 milesButton.setToggleGroup(radioGroup);
55 feetButton.setToggleGroup(radioGroup);
56 inchesButton.setToggleGroup(radioGroup);
57
58 // Create a Button to perform the conversion.
59 Button calcButton = new Button("Convert");
60
61 // Register the event handler.
62 calcButton.setOnAction(new CalcButtonHandler());
63
64 // Create an empty Label to display the result.
65 resultLabel = new Label();
66
67 // Put the promptLabel and the kiloTextField in an HBox.
```

```
68 HBox promptHBox = new HBox(10, promptLabel, kiloTextField);
69
70 // Put the RadioButtons in an HBox.
71 HBox radioHBox = new HBox(20, milesButton, feetButton,
72 inchesButton);
73
74 // Put everything in a VBox.
75 VBox mainVBox = new VBox(10, promptHBox, radioHBox, calcBut-
76 resultLabel);
77
78 // Set the VBox's alignment to center.
79 mainVBox.setAlignment(Pos.CENTER);
80
81 // Set the VBox's padding to 10 pixels.
82 mainVBox.setPadding(new Insets(10));
83
84 // Create a Scene.
85 Scene scene = new Scene(mainVBox);
86
87 // Add the Scene to the Stage.
88 primaryStage.setScene(scene);
89
90 // Set the stage title.
91 primaryStage.setTitle("Metric Converter");
92
93 // Show the window.
94 primaryStage.show();
95 }
96
97 /**
98 * Event handler class for calcButton
99 */
100
101 class CalcButtonHandler implements EventHandler<ActionEvent>
102 {
103 @Override
104 public void handle(ActionEvent event)
105 {
106 // Constants for the conversion factors.
107 final double MILES_CONVERSION = 0.6214;
108 final double FEET_CONVERSION = 3281.0;
109 final double INCHES_CONVERSION = 39370.0;
110
111 // Variable to hold the result
112 double result = 0;
113
114 // Get the kilometers.
```

```

115 double kilometers = Double.parseDouble(kiloTextField.get
116
117 // Perform the selected conversion.
118 if (milesButton.isSelected())
119 result = kilometers * MILES_CONVERSION;
120
121 if (feetButton.isSelected())
122 result = kilometers * FEET_CONVERSION;
123
124 if (inchesButton.isSelected())
125 result = kilometers * INCHES_CONVERSION;
126
127 // Display the results.
128 resultLabel.setText(String.format("%,.2f", result));
129 }
130 }
131 }
```

Let's take a closer look at the application:

- Lines 1 through 14 have the necessary JavaFX import statements.
- Lines 23 through 27 declare kiloTextField, resultLabel milesButton, feetButton, and inchesButton as fields in the MetricConverter class. The reason we are declaring them as fields (instead of declaring them as local variables in the start method) is because the event handler object needs to access them.
- Line 39 creates a Label control named promptLabel, to prompt the user to enter a distance in kilometers.
- Line 42 creates a TextField control and assigns it to the kiloTextField field.
- Lines 45 through 47 create the RadioButton controls and assign them to the milesButton, feetButton, and inchesButton fields.
- Line 50 causes the milesButton control to be initially selected.
- Line 53 creates a ToggleGroup object named radioGroup, and lines 54 through 56 add each of the RadioButton controls to the ToggleGroup.

- Line 59 creates a `Button` control named `calcButton`.
- Line 62 calls the `calcButton` control's `setOnAction` method to register an instance of the `CalcButtonHandler` class as an event handler with the `Button`.
- Line 65 creates an empty `Label` control and assigns it to the `resultLabel` field.
- Line 68 puts the `promptLabel` and `kiloTextField` controls in an `HBox` named `promptHBox`, with 10 pixels of spacing between them.
- Lines 71 and 72 put the `RadioButton` controls in an `HBox` named `radioHBox`, with 20 pixels of spacing between them.
- Lines 75 and 76 put the `promptHBox`, `radioHBox`, `calcButton`, and `resultLabel` controls in a `VBox`, with 10 pixels of spacing between them.
- Lines 79 and 82 set the `vBox`'s alignment to center, and the padding to 10 pixels.
- Lines 85 through 94 perform all the necessary steps to add the `VBox` to a scene, and add the scene to the stage. A title is displayed in the window, and the window is displayed.
- The event handler appears in lines 101 through 130. It is an inner class named `CalcButtonHandler`. The `handle` method appears in lines 104 through 129.
  - Lines 107 through 109 declare named constants for the necessary conversion factors.
  - Line 112 declares a local variable named `result`, to hold the result of the conversion.
  - Line 115 gets the value that the user entered into the `kiloTextField` control, converts it to a `Double`, and assigns it to the `kilometers` variable.

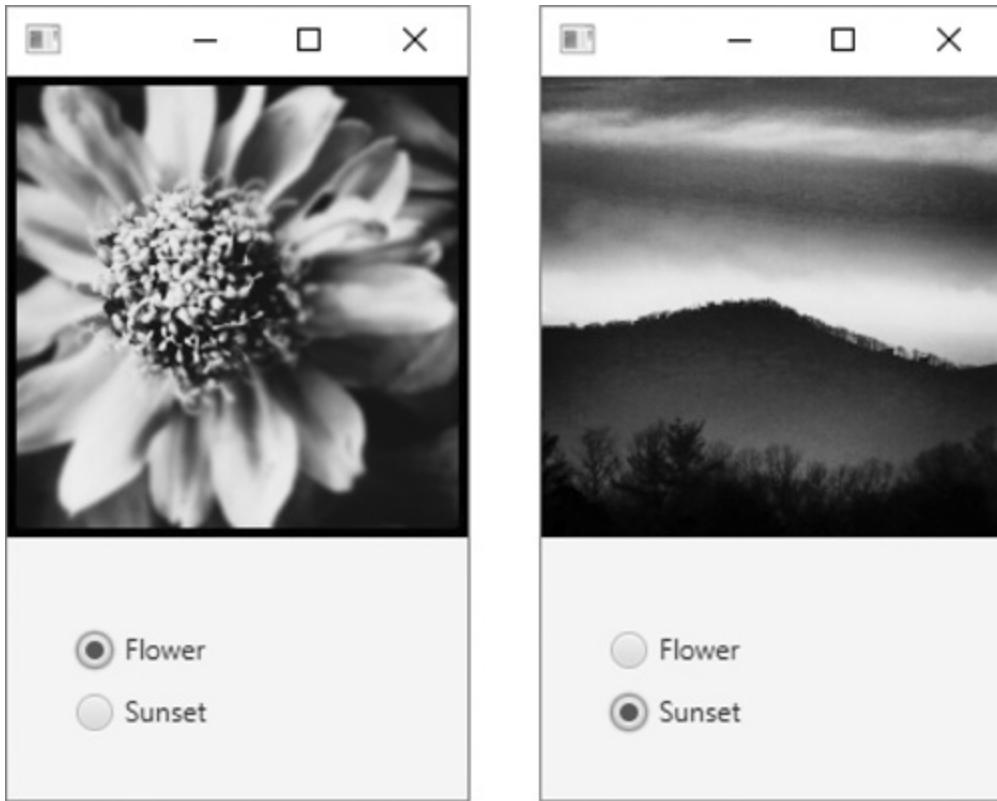
- The `if` statements in lines 118 through 125 determine which `RadioButton` is selected, and performs the chosen conversion operation.
- Line 128 displays the results in the `resultLabel` control.

## Responding to RadioButton Clicks

Sometimes, you merely want to know if a `RadioButton` is selected. When that is the case, you call the `RadioButton` control's `isSelected` method. Other times, however, you want an action to take place immediately when the user clicks a `RadioButton`. Just like `Button` controls, `RadioButton` controls generate an `ActionEvent` when they are clicked. If you want to respond to a `RadioButton`'s action event, you have to register an event handler with the control. The process of creating and registering an event handler for a `RadioButton` is the same as the process of creating and registering an event handler for a `Button`.

[Code Listing 12-15](#) shows an example. When the program runs, it displays the window shown on the left in [Figure 12-9](#). Initially the image of a flower is displayed. When the user clicks the *Sunset* `RadioButton`, the image immediately changes as shown in the window on the right. The user can click the `RadioButton` controls to display either the flower or the sunset.

## Figure 12-9 The RadioButtonEvent application



## Code Listing 12-15 (RadioButtonEvent.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.image.Image;
9 import javafx.scene.image.ImageView;
10 import javafx.scene.control.RadioButton;
11 import javafx.scene.control.ToggleGroup;
12 import javafx.event.EventHandler;
13 import javafx.event.ActionEvent;
14
15 /**
16 * A RadioButton ActionEvent Demo
17 */
18
```

```
19 public class RadioButtonEvent extends Application
20 {
21 public static void main(String[] args)
22 {
23 // Launch the application.
24 launch(args);
25 }
26
27 @Override
28 public void start(Stage primaryStage)
29 {
30 // Create two Image objects.
31 Image flowerImage = new Image("file:Flower.jpg");
32 Image sunsetImage = new Image("file:Sunset.jpg");
33
34 // Create an ImageView object.
35 ImageView imageView = new ImageView(flowerImage);
36
37 // Resize the ImageView, preserving its aspect ratio.
38 imageView.setFitWidth(200);
39 imageView.setPreserveRatio(true);
40
41 // Put the ImageView in an HBox.
42 HBox imageHBox = new HBox(imageView);
43
44 // Center the HBox contents.
45 imageHBox.setAlignment(Pos.CENTER);
46
47 // Create the RadioButtons.
48 RadioButton flowerRadio = new RadioButton("Flower");
49 RadioButton sunsetRadio = new RadioButton("Sunset");
50
51 // Select the flowerRadio control.
52 flowerRadio.setSelected(true);
53
54 // Add the RadioButtons to a ToggleGroup.
55 ToggleGroup radioGroup = new ToggleGroup();
56 flowerRadio.setToggleGroup(radioGroup);
57 sunsetRadio.setToggleGroup(radioGroup);
58
59 // Register an ActionEvent handler for the flowerRadio.
60 flowerRadio.setOnAction(event ->
61 {
62 imageView.setImage(flowerImage);
63 });
64
65 // Register an event handler for the sunsetRadio.
66 sunsetRadio.setOnAction(event ->
```

```

67 {
68 imageView.setImage(sunsetImage);
69 });
70
71 // Add the RadioButtons to a VBox.
72 VBox radioVBox = new VBox(10, flowerRadio, sunsetRadio);
73
74 // Give the radioVBox some padding.
75 radioVBox.setPadding(new Insets(30));
76
77 // Add everything to a VBox.
78 VBox mainVBox = new VBox(10, imageHBox, radioVBox);
79
80 // Create a Scene with the HBox as its root node.
81 Scene scene = new Scene(mainVBox);
82
83 // Add the Scene to the Stage.
84 primaryStage.setScene(scene);
85
86 // Show the window.
87 primaryStage.show();
88 }
89 }
```

Let's take a closer look at the program:

- Lines 1 through 13 have the necessary JavaFX import statements.
- Lines 31 and 32 load the images.
- Line 35 creates an `ImageView` object, initially displaying the flower.
- In lines 38 through 45, the `ImageView` object is resized and placed in an `HBox` with center alignment.
- Lines 48 and 49 create two `RadioButton` controls named `flowerRadio` and `sunsetRadio`, and line 52 selects the `flowerRadio` control. Lines 55 through 57 add the `RadioButton` controls to a `ToggleGroup`.
- Lines 60 through 63 use a lambda expression to create and register an event handler for the `flowerRadio` control. When the event handler executes, it changes the image that is displayed by the `imageView` control to the flower image.

- Lines 66 through 69 use a lambda expression to create and register an event handler for the `sunsetRadio` control. When the event handler executes, it changes the image that is displayed by the `imageView` control to the sunset image.
- Line 72 adds the `RadioButton` controls to a `VBox`, and line 75 sets the padding for the `VBox`.
- Lines 78 through 87 perform all the necessary steps to add the controls to layout containers, create a scene, add the scene to the stage, and display the window.



## Checkpoint

1. 12.12 How do you create a mutually exclusive relationship between `RadioButton` controls?
2. 12.13 How do you determine in code whether a `RadioButton` is selected?
3. 12.14 In code, how do you make a `RadioButton` appear selected?
4. 12.15 What type of event do `RadioButton` controls generate when they are clicked?

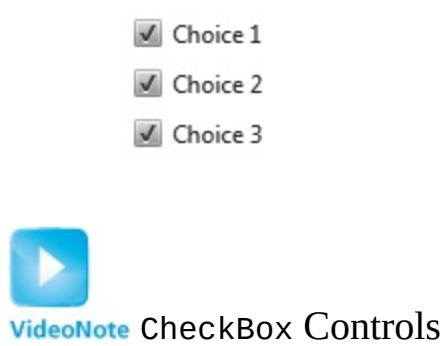
## 12.3 CheckBox Controls

### Concept:

*CheckBox controls, which may appear alone or in groups, allow the user to make yes/no or on/off selections.*

A CheckBox appears as a small box with a label appearing next to it. [Figure 12-10](#) shows three CheckBox controls.

### Figure 12-10 CheckBox controls



**VideoNote** CheckBox Controls

Like RadioButtons, CheckBox controls may be selected or deselected at runtime. When a CheckBox is selected, a small check mark appears inside the box. Although CheckBox controls are often displayed in groups, they are not usually grouped in a ToggleGroup like RadioButtons. This is because CheckBox controls are not normally used to make mutually exclusive selections. Instead, the user is allowed to select any or all of the CheckBox controls displayed within a group.

To create a CheckBox control, you will use the CheckBox class, which is in the javafx.scene.control package. Once you have written the necessary

import statement, you will create an instance of the CheckBox class. The following statement shows an example of creating a CheckBox object that displays the text *Macaroni*:

```
CheckBox check1 = new CheckBox("Macaroni");
```

Notice we pass the string we want displayed next to the CheckBox as an argument to the CheckBox class's constructor. Optionally, you can omit the string argument to create a CheckBox control with no text displayed next to it. Here is an example:

```
CheckBox check1 = new CheckBox();
```

## Determining in Code Whether a CheckBox Is Selected

As with RadioButton controls, you use the isSelected method to determine whether a CheckBox control is selected. The method returns a boolean value indicating whether the CheckBox is selected. If the CheckBox is selected, the method returns true. Otherwise, it returns false. In the following code, the check1 variable references a CheckBox. The if statement calls the isSelected method to determine whether the CheckBox is selected.

```
if (check1.isSelected())
{
 // Code here executes if the radio
 // button is selected.
}
```

## Selecting a CheckBox in Code

You can select a CheckBox in code with the CheckBox class's setSelected method. The setSelected method takes a boolean argument. If you pass true to the method, the CheckBox will be selected just as if the user had clicked it. (If you pass false as an argument to the setSelected method, the

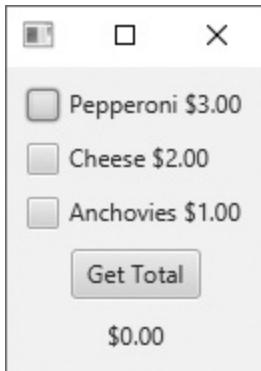
CheckBox will be deselected.)

In the following statement, the check1 variable references a CheckBox. When this statement executes, the check1 control will be selected.

```
check1.setSelected(true);
```

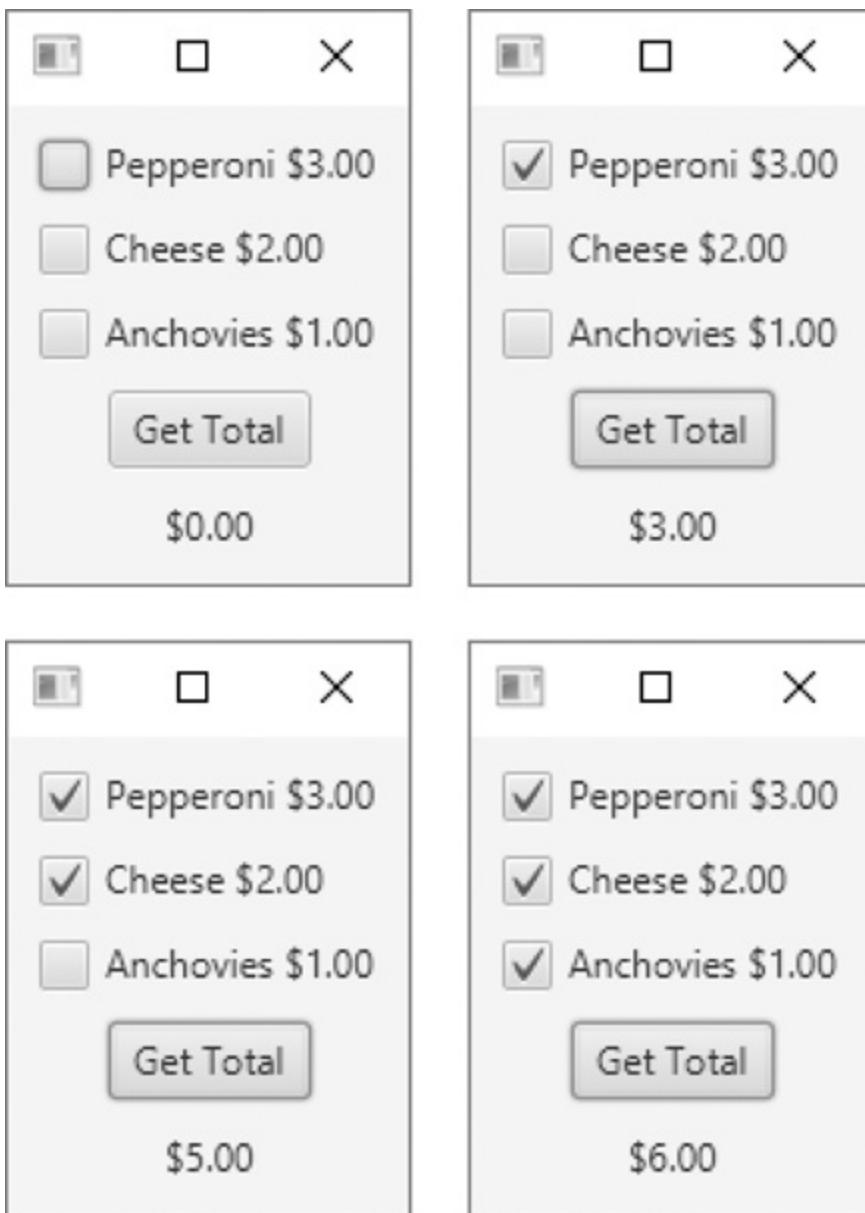
Let's look at an example. [Code Listing 12-16](#) shows the code for the PizzaToppings application, and [Figure 12-11](#) shows how the application's window initially appears. The user checks the pizza topping items, then clicks the Button control to see the total cost of the selected items.

## Figure 12-11 The PizzaToppings application



[Figure 12-12](#) shows four different screenshots of the application running. Each screenshot shows the application after the user has made selections and clicked the Button control. The total of the selected items is displayed in the Label.

## Figure 12-12 The PizzaToppings application running



[Figure 12-12 Full Alternative Text](#)

## Code Listing 12-16 (**PizzaToppings.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
```

```
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.CheckBox;
10 import javafx.scene.control.Button;
11 import javafx.event.EventHandler;
12 import javafx.event.ActionEvent;
13
14 /**
15 * CheckBox Demo application
16 */
17
18 public class PizzaToppings extends Application
19 {
20 // Fields
21 CheckBox pepperoniCheckBox;
22 CheckBox cheeseCheckBox;
23 CheckBox anchoviesCheckBox;
24 Label totalLabel;
25
26 public static void main(String[] args)
27 {
28 // Launch the application.
29 launch(args);
30 }
31
32 @Override
33 public void start(Stage primaryStage)
34 {
35 // Create the CheckBoxes.
36 pepperoniCheckBox = new CheckBox("Pepperoni $3.00");
37 cheeseCheckBox = new CheckBox("Cheese $2.00");
38 anchoviesCheckBox = new CheckBox("Anchovies $1.00");
39
40 // Create the Button control.
41 Button totalButton = new Button("Get Total");
42
43 // Register the event handler.
44 totalButton.setOnAction(new TotalButtonHandler());
45
46 // Create a Label for the total.
47 totalLabel = new Label("$0.00");
48
49 // Put the CheckBoxes in a VBox.
50 VBox checkBoxVBox = new VBox(10, pepperoniCheckBox,
51 cheeseCheckBox, anchoviesCheckBox);
52 }
53 }
```

```
53 // Create another VBox to use as the root node.
54 VBox mainVBox = new VBox(10, checkBoxVBox, totalButton,
55 totalLabel);
56
57 // Set the main VBox's alignment to center.
58 mainVBox.setAlignment(Pos.CENTER);
59
60 // Set the main VBox's padding to 10 pixels.
61 mainVBox.setPadding(new Insets(10));
62
63 // Create a Scene.
64 Scene scene = new Scene(mainVBox);
65
66 // Add the Scene to the Stage.
67 primaryStage.setScene(scene);
68
69 // Show the window.
70 primaryStage.show();
71 }
72
73 /**
74 * Event handler class for totalButton
75 */
76
77 class TotalButtonHandler implements EventHandler<ActionEvent>
78 {
79 @Override
80 public void handle(ActionEvent event)
81 {
82 // Variable to hold the result
83 double result = 0;
84
85 // Add up the toppings.
86 if (pepperoniCheckBox.isSelected())
87 result += 3.0;
88
89 if (cheeseCheckBox.isSelected())
90 result += 2.0;
91
92 if (anchoviesCheckBox.isSelected())
93 result += 1.0;
94
95 // Display the results.
96 totalLabel.setText(String.format("$%, .2f", result));
97 }
98 }
99 }
```

Let's take a closer look at the application:

- Lines 1 through 12 have the necessary JavaFX `import` statements.
- Lines 21 through 24 declare `pepperoniCheckBox`, `cheeseCheckBox`, `anchoviesCheckBox`, and `totalLabel` as fields in the `PizzaToppings` class. The reason we are declaring them as fields (instead of declaring them as local variables in the `start` method) is because the event handler object needs to access them.
- Lines 36 through 38 create the `CheckBox` controls, and line 41 creates the `Button` control.
- Line 44 registers an instance of the `TotalButtonHandler` class as an event handler for the `Button` control.
- Line 47 creates a `Label` control to display the total.
- Lines 50 and 51 put the `CheckBox` controls in a `VBox`.
- Lines 54 and 55 create another `VBox` to serve as the root node. All of the controls are added to this `VBox`. Lines 58 and 61 set the `VBox`'s alignment to center, and the padding to 10 pixels.
- Lines 64 through 70 perform all the necessary steps to add the root node to a scene, add the scene to the stage, and display the window.
- The event handler appears in lines 77 through 98. It is an inner class named `TotalButtonHandler`. The `handle` method appears in lines 80 through 97.
  - Line 83 declares a local variable named `result`, initialized to 0, to hold the sum of the selected toppings.
  - The `if` statements in lines 86 through 93 determine which of the `CheckBox` controls is selected, and adds the selected topping amounts to the `result` variable.
  - Line 96 displays the results in the `totalLabel` control.

# Responding to CheckBox Clicks

Just like Button and RadioButton controls, CheckBox controls generate an ActionEvent when they are clicked. If you want an action to take place immediately when the user clicks a CheckBox, you have to register an event handler with the control. The process of creating and registering an event handler for a CheckBox is the same as the process of creating and registering an event handler for a Button or a RadioButton.



## Checkpoint

1. 12.16 How do you determine in code whether a CheckBox is selected?
2. 12.17 In code, how do you make a CheckBox appear selected?
3. 12.18 What type of event do CheckBox controls generate when they are clicked?

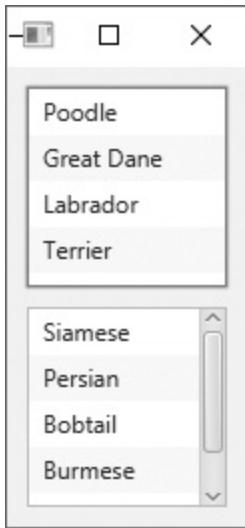
## 12.4 ListView Controls

### Concept:

*The ListView control displays a list of items and allows the user to select an item from the list.*

The ListView control displays a list of items and allows the user to select one or more items from the list. [Figure 12-13](#) shows an application's window with two ListView controls. Notice the topmost ListView does not have a scroll bar, but the bottom one does. A scroll bar automatically appears when a ListView contains more items than can be displayed in the space provided. In the figure, the top ListView has four items (Poodle, Great Dane, Labrador, and Terrier), and all items are displayed. The bottom ListView shows four items (Siamese, Persian, Bobtail, and Burmese), but because it has a scroll bar, we know there are more items in the ListView than those four.

### Figure 12-13 ListView examples



[Figure 12-13 Full Alternative Text](#)



[VideoNote](#) ListView Controls

You create a `ListView` control with the `ListView` class (which is in the `javafx.scene.control` package). The following code snippet shows an example of creating a `ListView` named `dogListView`:

```
ListView<String> dogListView = new ListView<>();
```

Notice at the beginning of the statement, the notation `<String>` appears immediately after the word `ListView`. This specifies that this particular `ListView` control can hold only `String` objects. If we try to store any other type of object in this `ListView`, an error will occur when we compile the program. Also notice the diamond operator `<>` appears after `ListView` at the end of the statement. (You might recall from [Chapter 7](#) that `ArrayList` objects are declared in a similar fashion.)

Once you have created a `ListView` control, you are ready to add items to it. The following statement shows how we can add strings to our `dogListView` control:

```
dogListView.getItems().addAll("Poodle", "Great Dane", "Labrador",
```



## Note:

The `ListView` control keeps its list of items in an `ObservableList` object. We discussed the `ObservableList` interface in [Chapter 11](#). The `ListView` class's `getItems()` method returns the `ObservableList` that holds the `ListView`'s items.

You can set the preferred size of the `ListView` control with the `setPreferredSize` method. You pass the preferred width and height as arguments. The following code snippet shows an example of creating a `ListView` control that is 120 pixels wide by 100 pixels high:

```
ListView<String> catListView = new ListView<>();
catListView.setPrefSize(120, 100);
catListView.getItems().addAll("Siamese", "Persian", "Bobtail",
 "Burmese", "Tabby");
```

## Retrieving the Selected Item

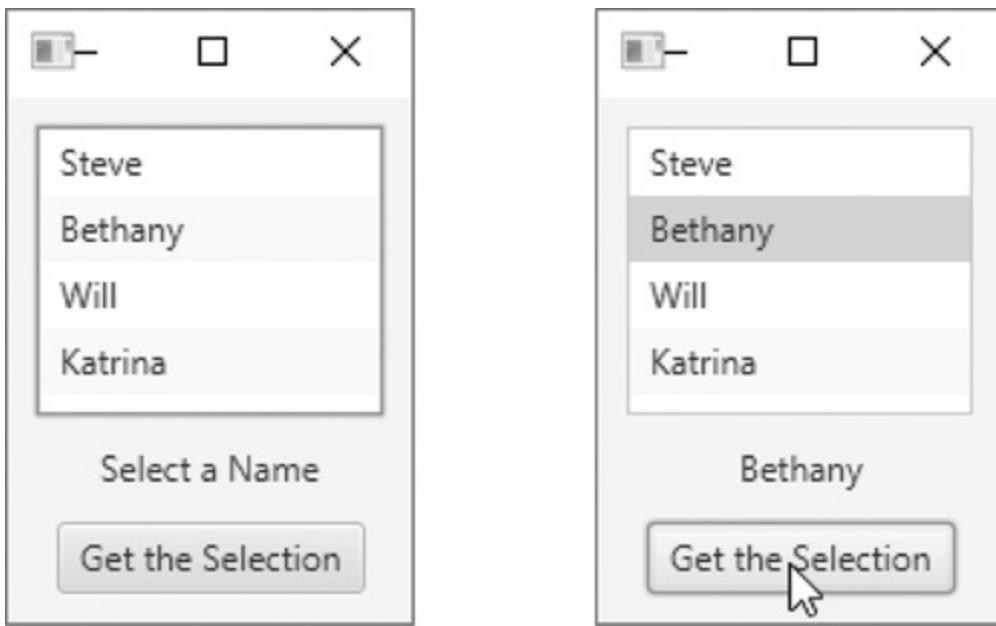
You can use the `ListView` class's `getSelectionModel().getSelectedItem()` method get the item that is currently selected. For example, assume we have a `ListView` control named `listView`. The following code gets the item that is currently selected in the control, and assigns it to a `String` named `selected`:

```
String selected = listView.getSelectionModel().getSelectedItem();
```

If no item is selected in the `ListView`, the method will return `null`. [Code Listing 12-17](#) shows a complete example. It displays a list of names in a `ListView` control. When the user clicks the `Button` control, the program gets the selected name and displays it in a `Label` control. [Figure 12-14](#) shows the program running.

## Figure 12-14 The

# **ListVieDemo1.java** **application**



[Figure 12-14 Full Alternative Text](#)

## **Code Listing 12-17** **(ListViewDemo1.java)**

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.scene.layout.VBox;
8 import javafx.geometry.Pos;
9 import javafx.geometry.Insets;
10
11 public class ListViewDemo1 extends Application
12 {
13 public static void main(String[] args)
```

```

14 {
15 // Launch the application.
16 launch(args);
17 }
18
19 @Override
20 public void start(Stage primaryStage)
21 {
22 // Create a ListView of Strings.
23 ListView<String> listView = new ListView<>();
24 listView.setPrefSize(120, 100);
25 listView.getItems().addAll("Steve", "Bethany", "Will", "Kat"
26
27 // Create a Label to display the selection.
28 Label selectedNameLabel = new Label("Select a Name");
29
30 // Create a Button to get the selection.
31 Button getButton = new Button("Get the Selection");
32
33 // Create an event handler for the Button.
34 getButton.setOnAction(event ->
35 {
36 // Get the selected name.
37 String selected = listView.getSelectionModel().getSelecte
38
39 // Display the selected name in the Label.
40 selectedNameLabel.setText(selected);
41 });
42
43 // Add the controls to a VBox.
44 VBox vbox = new VBox(10, listView, selectedNameLabel, getBu
45 vbox.setPadding(new Insets(10));
46 vbox.setAlignment(Pos.CENTER);
47
48 // Create a Scene and display it.
49 Scene scene = new Scene(vbox);
50 primaryStage.setScene(scene);
51 primaryStage.show();
52 }
53 }
```

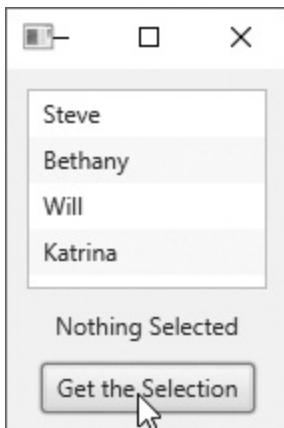
# Retrieving the Index of the Selected Item

Internally, each item in a `ListView` control is assigned an index that marks the item's position. The first item (which is the item stored at the top of the list) has the index 0, the second item has the index 1, and so forth. You can use the `ListView` class's `getSelectionModel().getSelectedIndex()` method get the index of the item that is currently selected. For example, assume we have a `ListView` control named `listView`. The following code gets the index of the item that is currently selected in the control, and assigns it to an `int` variable named `index`:

```
int index = listView.getSelectionModel().getSelectedIndex();
```

If no item is selected in the `ListView`, the method will return -1. You can use this to determine whether the user has selected an item. For example, [Code Listing 12-18](#) is a modification of the program in [Code Listing 12-17](#). In this version, the program displays the message “Nothing Selected” if the user clicks the `Button` control before he or she has selected an item in the `ListView` control. An example of this output is shown in [Figure 12-15](#).

## Figure 12-15 The ListViewDemo2.java application



[Figure 12-15 Full Alternative Text](#)

# Code Listing 12-18 (ListViewDemo2.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.scene.layout.VBox;
8 import javafx.geometry.Pos;
9 import javafx.geometry.Insets;
10
11 public class ListViewDemo2 extends Application
12 {
13 public static void main(String[] args)
14 {
15 // Launch the application.
16 launch(args);
17 }
18
19 @Override
20 public void start(Stage primaryStage)
21 {
22 // Create a ListView of Strings.
23 ListView<String> listView = new ListView<>();
24 listView.setPrefSize(120, 100);
25 listView.getItems().addAll("Steve", "Bethany", "Will", "Kat"
26
27 // Create a Label to display the selection.
28 Label selectedNameLabel = new Label("Select a Name");
29
30 // Create a Button to get the selection.
31 Button getButton = new Button("Get the Selection");
32
33 // Create an event handler for the Button.
34 getButton.setOnAction(event ->
35 {
36 // Determine whether an item is selected.
37 if (listView.getSelectionModel().getSelectedIndex() != -1
38 {
39 // Get the selected name.
40 String selected =
41 listView.getSelectionModel().getSelectedItem());
```

```

42 // Display the selected name in the Label.
43 selectedNameLabel.setText(selected);
44 }
45 else
46 {
47 selectedNameLabel.setText("Nothing Selected");
48 }
49 }
50);
51
52 // Add the controls to a VBox.
53 VBox vbox = new VBox(10, listView, selectedNameLabel, getBu
54 vbox.setPadding(new Insets(10));
55 vbox.setAlignment(Pos.CENTER);
56
57 // Create a Scene and display it.
58 Scene scene = new Scene(vbox);
59 primaryStage.setScene(scene);
60 primaryStage.show();
61 }
62 }
```

Let's take a closer look at the program. In the `Button` control's event handler, the `if` statement that begins in line 37 gets the index of the selected item. If that value is *not* -1, then an item has been selected, and the selected item is displayed (in line 44). However, if the selected item index is -1, then nothing is selected in the `Listview`, and the program jumps to the `else` clause in line 46.

## Responding to Item Selection with an Event Handler

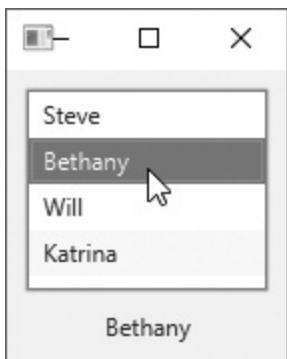
When the user selects an item in a `Listview`, a *change event* occurs. If you want the program to immediately perform an action when the user selects an item in a `Listview`, you can write an event handler that responds to the change event. Assuming `listView` is the name of a `Listview` control, the following code snippet shows how to use a lambda expression to register such an event handler:

```
listView.getSelectionModel().selectedItemProperty().addListener(e
```

```
{
 // Write event handling code here ...});
```

For example, [Code Listing 12-19](#) is another modification of the program in [Code Listing 12-17](#). This version of the application does not have a Button control. Instead, the selected item is displayed immediately after the user makes his or her selection. An example of the program's output is shown in [Figure 12-16](#).

## Figure 12-16 The ListViewDemo3.java application



[Figure 12-16 Full Alternative Text](#)

## Code Listing 12-19 (ListViewDemo3.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.Label;
6 import javafx.scene.layout.VBox;
7 import javafx.geometry.Pos;
```

```

8 import javafx.geometry.Insets;
9
10 public class ListViewDemo3 extends Application
11 {
12 public static void main(String[] args)
13 {
14 // Launch the application.
15 launch(args);
16 }
17
18 @Override
19 public void start(Stage primaryStage)
20 {
21 // Create a ListView of Strings.
22 ListView<String> listView = new ListView<>();
23 listView.setPrefSize(120, 100);
24 listView.getItems().addAll("Steve", "Bethany", "Will", "Kat"
25
26 // Create a Label to display the selection.
27 Label selectedNameLabel = new Label("Select a Name");
28
29 // Create an event handler for the ListView control.
30 listView.getSelectionModel().selectedItemProperty().addListener(
31 {
32 // Get the selected name.
33 String selected = listView.getSelectionModel().getSelected
34
35 // Display the selected name in the Label.
36 selectedNameLabel.setText(selected);
37 });
38
39 // Add the controls to a VBox.
40 VBox vbox = new VBox(10, listView, selectedNameLabel);
41 vbox.setPadding(new Insets(10));
42 vbox.setAlignment(Pos.CENTER);
43
44 // Create a Scene and display it.
45 Scene scene = new Scene(vbox);
46 primaryStage.setScene(scene);
47 primaryStage.show();
48 }
49 }

```

## Adding Items versus Setting Items

The `ListView` control's `getItems().addAll()` method adds items to the `ListView`. If the `ListView` already contains items, the `getItems().addAll()` method will not erase the existing items, but will add the new items to the existing list. As a result, the `ListView` will contain the old items plus the new items.

Sometimes this is the behavior you want, but in other situations, you might want to replace the existing contents of a `ListView` with an entirely new list of items. When that is the case, you can use the `getItems().setAll()` method. The `getItems().setAll()` method replaces a `ListView` control's existing items with a new list of items.

The following code snippet demonstrates how to create a `ListView` containing one list of items, then replace those items with another list:

```
// Create a ListView.
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday");
// Replace the existing items with a new list.
listView.getItems().setAll("Thursday", "Friday", "Saturday");
```

After this code executes, the `listView` control will contain the items “Thursday”, “Friday”, and “Saturday”.

## Initializing a `ListView` with an Array or an `ArrayList`

In some situations, you might have an array or an `ArrayList` containing items you want to display in a `ListView` control. When this is the case, you can convert the array or `ArrayList` to an `ObservableList`, then pass the `ObservableList` as an argument to the `ListView` class's constructor.

The `FXCollections` class (which is in the `javafx.collections` package) has a static method named `observableArrayList` that takes an array or an `ArrayList` as an argument, and returns an `ObservableList` containing the same items. The following code shows an example of creating a `ListView`

control that displays the items in a String array:

```
// Create a String array.
String[] strArray = {"Monday", "Tuesday", "Wednesday"};
// Convert the String array to an ObservableList.
ObservableList<String> strList =
 FXCollections.observableArrayList(strArray);
// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

The following code snippet shows the same example, but this time converting an ArrayList to an ObservableList:

```
// Create an ArrayList of Strings.
ArrayList<String> strArrayList = new ArrayList<>();
strArrayList.add("Monday");
strArrayList.add("Tuesday");
strArrayList.add("Wednesday");
// Convert the ArrayList to an ObservableList.
ObservableList<String> strList =
 FXCollections.observableArrayList(strArrayList);
// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

You can also populate a ListView by passing an ObservableList to the ListView class's getItems().addAll() and getItems().setAll() methods. Here is an example that uses the getItems().addAll() method to populate a ListView with the items in a String array:

```
// Create a String array.
String[] strArray = {"Monday", "Tuesday", "Wednesday"};
// Convert the String array to an ObservableList.
ObservableList<String> strList =
 FXCollections.observableArrayList(strArray);
// Create a ListView control.
ListView<String> listView = new ListView<>();
// Populate the ListView control:
listView.getItems().addAll(strList);
```

Here is an example that uses the getItems().setAll() method to populate a ListView with the items in an ArrayList of Strings:

```
// Create an ArrayList of Strings.
```

```
ArrayList<String> strArrayList = new ArrayList<>();
strArrayList.add("Monday");
strArrayList.add("Tuesday");
strArrayList.add("Wednesday");
// Convert the ArrayList to an ObservableList.
ObservableList<String> strList =
 FXCollections.observableArrayList(strArrayList);
// Create a ListView control.
ListView<String> listView = new ListView<>();
// Populate the ListView control:
listView.getItems().addAll(strList);
```

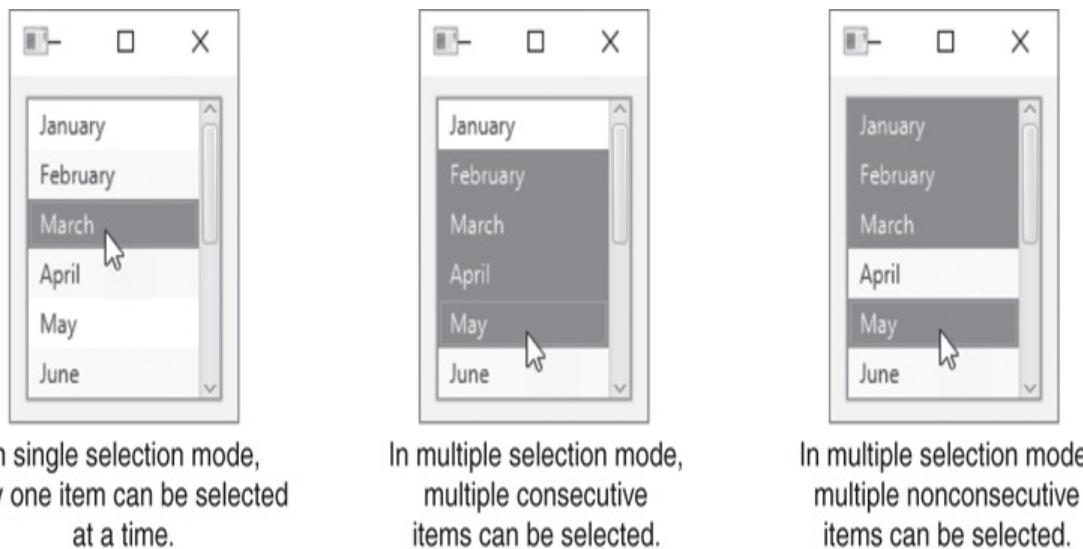
## Selection Modes

The `ListView` control can operate in either of the following selection modes:

- Single Selection Mode. In this mode, only one item can be selected at a time. When an item is selected, any other item that is currently selected is deselected. This is the default selection mode
- Multiple Interval Selection Mode. In this mode, multiple items may be selected. To select multiple items that are consecutive, the user clicks the first item, holds down the Shift key on the keyboard, then clicks the last item. To select items that are not consecutive, the user holds down the Ctrl key on the keyboard (or the Command key on a Mac keyboard), then clicks each item.

[Figure 12-17](#) shows examples of a `ListView` control in single selection mode and multiple selection mode.

## Figure 12-17 Selection modes



[Figure 12-17 Full Alternative Text](#)

By default, the `ListView` control is in single selection mode. You change the selection mode with

the control's `getSelectionModel().setSelectionMode()` method. The method accepts one of the following enum constants:

- `SelectionMode.MULTIPLE`
- `SelectionMode.SINGLE`

Note the `SelectioMode` enum is in the `javafx.scene.control` package. Assuming that `listView` is the name of a `ListView` control, the following code shows how to change the control to multiple selection mode:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULTI
```

## Retrieving Multiple Selected Items

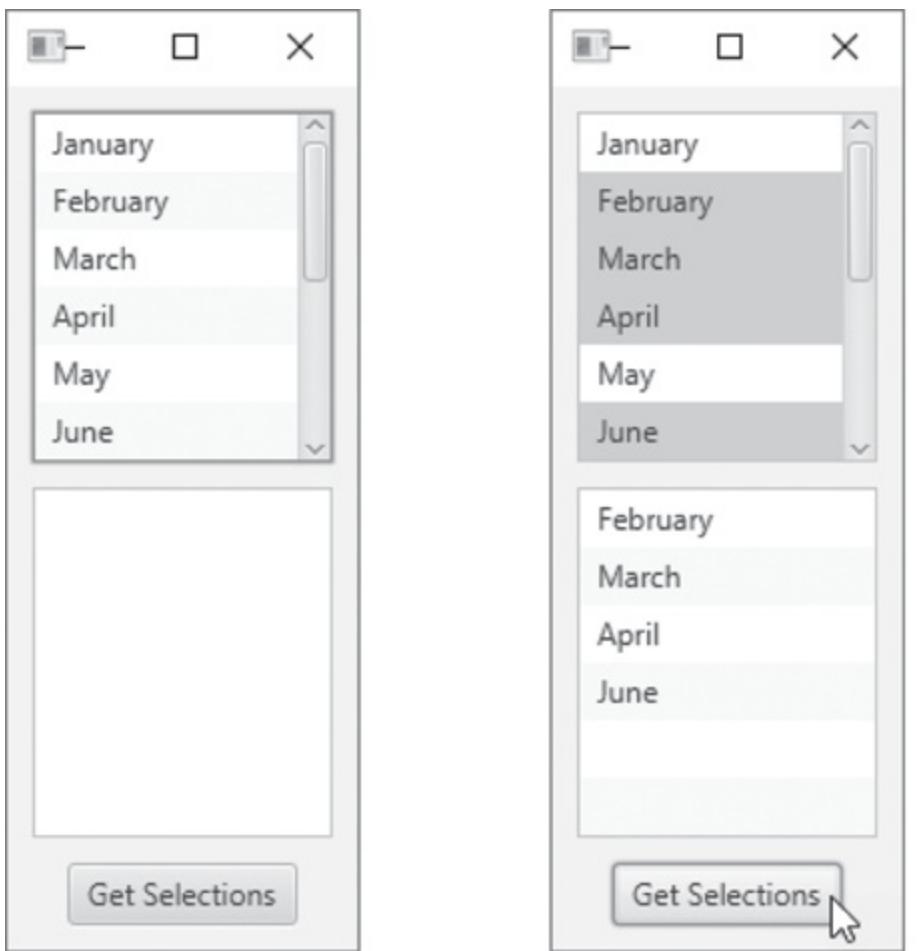
When a `ListView` control is in multiple selection mode, the user can select more than one item. The `getSelectionModel().getSelectedItem()` method will return only the last item selected. Likewise, the `getSelectionModel().getSelectedIndex()` method will return only the

index of the last item selected. To get all of the selected items and their indices, you need to use the following methods:

- `getSelectionModel().getSelectedItems()`—This method returns a read-only `ObservableList` of the selected items
- `getSelectionModel().getSelectedIndices()`—This method returns a read-only `ObservableList` of the integer indices of the selected items

[Code Listing 12-20](#) demonstrates how to retrieve multiple selected items. The program displays two `ListView` controls, named `listView1` and `listView2`. The user selects items from `listView1` and then, when the user clicks the `Button` control, the selected items are displayed in `listView2`. [Figure 12-18](#) shows an example of the application as it is running. The image on the left shows the application's window as it initially appears, and the image on the right shows the window after the user has selected several items from `listView1` and clicked the `Button` control.

## Figure 12-18 The ListViewDemo4.java application



The window as it initially appears

The window after the user has selected four items and clicked the button

[Figure 12-18 Full Alternative Text](#)

## Code Listing 12-20 (ListViewDemo4.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.SelectionMode;
6 import javafx.collections.ObservableList;
7 import javafx.scene.control.Label;
```

```
8 import javafx.scene.control.Button;
9 import javafx.scene.layout.VBox;
10 import javafx.geometry.Pos;
11 import javafx.geometry.Insets;
12
13 public class ListViewDemo4 extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Constants for the ListView sizes
25 final double WIDTH = 120, HEIGHT = 140;
26
27 // Create a ListView of the names of the months.
28 ListView<String> listView1 = new ListView<>();
29 listView1.setPrefSize(WIDTH, HEIGHT);
30 listView1.getSelectionModel().setSelectionMode(
31 SelectionMode.MULTIPLE);
32 listView1.getItems().addAll(
33 "January", "February", "March", "April", "May",
34 "June", "July", "August", "September", "October",
35 "November", "December");
36
37 // Create an empty ListView to show the selections.
38 ListView<String> listView2 = new ListView<>();
39 listView2.setPrefSize(WIDTH, HEIGHT);
40
41 // Create a Button to get the selections.
42 Button getButton = new Button("Get Selections");
43
44 // Register an event handler for the Button.
45 getButton.setOnAction(event ->
46 {
47 // Get the ObservableList of selected items.
48 ObservableList<String> selections =
49 listView1.getSelectionModel().getSelectedItems();
50
51 // Add the selections to the 2nd ListView.
52 listView2.getItems().setAll(selections);
53 });
54
55 // Add the controls to a VBox.
```

```

56 VBox vbox = new VBox(10, listView1,
57 listView2, getButton);
58 vbox.setPadding(new Insets(10));
59 vbox.setAlignment(Pos.CENTER);
60
61 // Create a Scene and display it.
62 Scene scene = new Scene(vbox);
63 primaryStage.setScene(scene);
64 primaryStage.show();
65 }
66 }
```

Let's take a closer look at the program:

- Line 25 declares constants for the `ListView` dimensions.
- Lines 28 through 35 create the `listView1` control, set its size, set its selection mode to `SelectionMode.MULTIPLE`, and populate it with the names of the months.
- Lines 38 and 39 create the `listView2` control and set its size.
- Line 42 creates a `Button` control, and lines 45 through 53 register an event handler for the `Button`.
  - Lines 48 and 49 get the `ObservableList` of items that are selected in the `listView1` control.
  - Line 52 displays the `ObservableList` in the `listView2` control.
- Lines 56 through 59 create a `VBox` and add all of the controls to it.
- Lines 62 through 64 create a scene, set the scene to the stage, and display the application's window.

## Working With the Elements of an `ObservableList`

An `ObservableList` has much in common with the `ArrayList` class that you

learned about in [Chapter 7](#). If you need to work with the individual elements in an `ObservableList`, everything that you have already learned about `ArrayLists` also applies to `Observables`. For example:

- You can use the enhanced `for` loop to iterate over an `ObservableList`.
- You can use the `get` method to get a specific element in an `ObservableList`.
- You can use the `size` method to get the number of elements in an `ObservableList`.
- You can use the `add` method to add an element to an `ObservableList` (as long as the `ObservableList` is not read-only).
- You can use the `remove` method to remove an element from an `ObservableList`.
- You can use the `set` method to replace an element in an `ObservableList`.

The following code snippet shows a different version of the `getButton` control's event handler in [Code Listing 12-20](#). In this version of the code, we are using an enhanced `for` loop in line 52 to iterate over the `ObservableList`, adding one item at a time to the `listView2` control (line 53):

```
44 // Register an event handler for the Button.
45 getButton.setOnAction(event ->
46 {
47 // Get the ObservableList of selected items.
48 ObservableList<String> selections =
49 listView1.getSelectionModel().getSelectedItems();
50
51 // Add the selections to the 2nd ListView.
52 for (String str : selections)
53 listView2.getItems().addAll(str);
54 });
```

The following code snippet shows yet another version of the `getButton` control's event handler in [Code Listing 12-20](#). In this version of the code, we are using a regular `for` loop in line 52 to iterate over the `ObservableList`,

adding one item at a time to the `listView2` control (line 53):

```
44 // Register an event handler for the Button.
45 getButton.setOnAction(event ->
46 {
47 // Get the ObservableList of selected items.
48 ObservableList<String> selections =
49 listView1.getSelectionModel().getSelectedItems();
50
51 // Add the selections to the 2nd ListView.
52 for (int i = 0; i < selections.size(); i++)
53 listView2.getItems().addAll(selections.get(i));
54 });
```

Obviously, neither of these approaches are better than the original approach shown in [Code Listing 12-20](#), but these examples illustrate how you can iterate over an `ObservableList` and work with its individual elements.

## Converting an `ObservableList` to an Array

Sometimes, you might need to get the `ObservableList` of selected items from a `ListView` control, then convert that `ObservableList` to an array. `ObservableList` objects have a method named `toArray` that returns an array containing the items in the list. Before the `toArray` method can convert the `ObservableList` to an array, however, it must know the data type of the array. You specify the data type by passing an empty array of the correct data type as an argument to the `toArray` method. Let's look at an example. In the following code snippet, assume `listView` is a `ListView` control containing strings:

```
// Get the selected items as an ObservableList.
ObservableList<String> selections =
 listView.getSelectionModel().getSelectedItems();
// Convert the ObservableList to an array of Strings.
String[] itemArray = selections.toArray(new String[0]);
```

In this code, the first statement gets an `ObservableList` containing the selected items in the `listView` control. The second statement converts the

`ObservableList` to a `String` array, and assigns the array to the `itemArray` variable. The empty `String` array we are passing as an argument to the `toArray` method tells the method to return a `String` array.

## Using Code to Select an Item in a ListView

When you create a `ListView` control, none of the items in the `ListView` are initially selected by default. In code, you can use any of the methods listed in [Table 12-5](#) to select an item or items in a `ListView` control.

**Table 12-5 ListView methods to select items**

| Method                                                   | Description                                                                                        |
|----------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| <code>getSelectionModel().selectFirst()</code>           | Selects the first item in the <code>ListView</code> .                                              |
| <code>getSelectionModel().selectLast()</code>            | Selects the last item in the <code>ListView</code> .                                               |
| <code>getSelectionModel().selectAll()</code>             | Selects all of the items in the <code>ListView</code> .                                            |
| <code>getSelectionModel().selectIndices(index...)</code> | Pass one or more <code>int</code> indices as arguments, and this method selects each corresponding |

```
getSelectionModel().selectRange(start, end)
```

item in the ListView. This method selects a range of items in the ListView, beginning at the *start* index, and ending at the *end* index. Both arguments are ints.

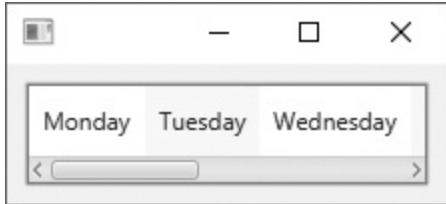
For example, the following code creates a ListView and selects the first item:

```
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday");
listView.getSelectionModel().selectFirst();
```

## ListView Orientation

By default, ListView controls are vertically oriented. You can set the orientation to either vertical or horizontal with the ListView class's `setOrientation` method. When you call the method, you pass it one of the following enum constants: `Orientation.VERTICAL` or `Orientation.HORIZONTAL`. (The `Orientation` enum is in the `javafx.geometry` package.) The following code snippet shows how to create a ListView control and set its orientation to horizontal. [Figure 12-19](#) shows an example of how the ListView will appear.

## Figure 12-19 A horizontal ListView



```
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday",
 "Thursday", "Friday", "Saturday",
 "Sunday");
listView.setOrientation(Orientation.HORIZONTAL);
listView.setPrefSize(200, 50);
```

## Creating ListViews of Objects Other Than String

All of the examples we have discussed so far use `ListViews` of `Strings`. You can store other types of objects in a `Listview` as well. The following code snippet creates a `Listview` that displays the integers 1 through 5:

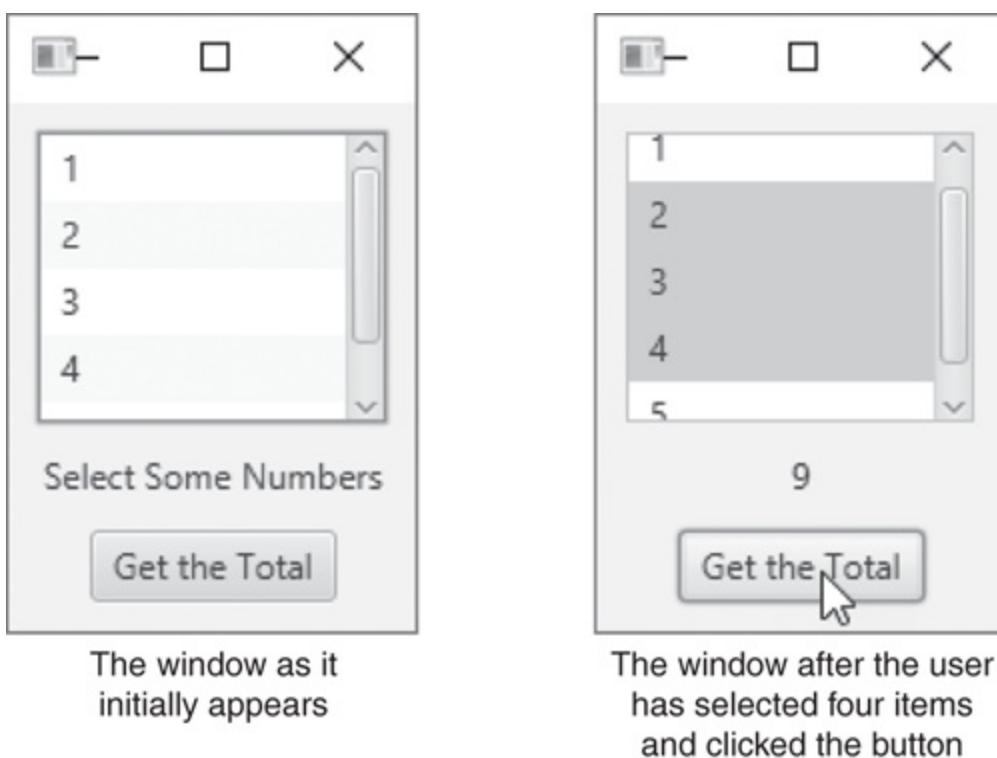
```
ListView<Integer> listView = new ListView<>();
listView.getItems().addAll(1, 2, 3, 4, 5);
```

Notice in the first statement, the data type of the `Listview` is `ListView<Integer>`. If you want to display values of one of the primitive types, such as `int`, `double`, `float`, and so on, you must specify the type's corresponding wrapper class in the `Listview` declaration. For example, the following code snippet creates a `Listview` that displays values of the `double` type:

```
ListView<Double> listView = new ListView<>();
listView.getItems().addAll(1.0, 2.0, 3.0, 4.0, 5.0);
```

[Code Listing 12-21](#) shows a complete example. As shown in [Figure 12-20](#), this program creates a `Listview` of the integer values 1 through 5. The `Listview` is in multiple selection mode, so the user can select multiple values. When the user clicks the `Button` control, the program calculates the sum of the selected integers, and displays the sum in the `Label` control.

# Figure 12-20 The ListViewDemo5.java application



[Figure 12-20 Full Alternative Text](#)

## Code Listing 12-21 (ListViewDemo5.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.SelectionMode;
6 import javafx.scene.control.Label;
7 import javafx.scene.control.Button;
```

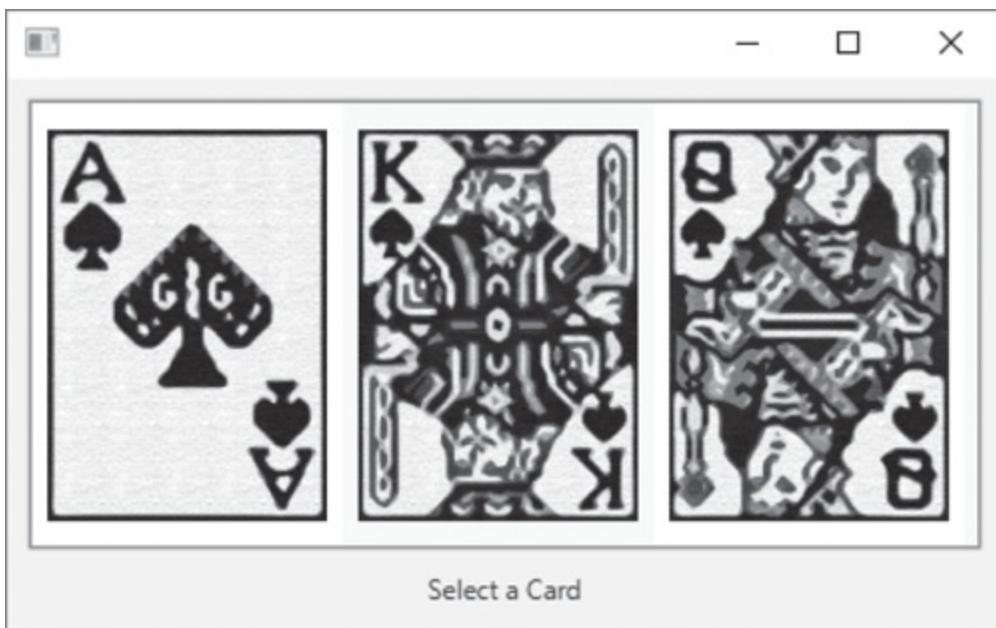
```
8 import javafx.collections.ObservableList;
9 import javafx.scene.layout.VBox;
10 import javafx.geometry.Pos;
11 import javafx.geometry.Insets;
12
13 public class ListViewDemo5 extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Constants for the ListView width and height
25 final double WIDTH = 120.0, HEIGHT = 100.0;
26
27 // Create a ListView of Integers.
28 ListView<Integer> listView = new ListView<>();
29 listView.setPrefSize(WIDTH, HEIGHT);
30 listView.getItems().addAll(1, 2, 3, 4, 5);
31 listView.getSelectionModel().setSelectionMode(
32 SelectionMode.MULTIPLE);
33
34 // Create a Label to display the selection.
35 Label outputLabel = new Label("Select Some Numbers");
36
37 // Create a Button to get the selection.
38 Button totalButton = new Button("Get the Total");
39
40 // Create an event handler for the Button.
41 totalButton.setOnAction(event ->
42 {
43 // Determine whether an item is selected.
44 if (listView.getSelectionModel().getSelectedIndex() != -1)
45 {
46 // Get the ObservableList of selected items.
47 ObservableList<Integer> selections =
48 listView.getSelectionModel().getSelectedItems();
49
50 // Accumulator variable
51 int total = 0;
52
53 // Get the sum of the numbers.
54 for (int num : selections)
```

```

55 total += num;
56
57 // Display the total in the Label.
58 outputLabel.setText(Integer.toString(total));
59 }
60 else
61 {
62 outputLabel.setText("Nothing Selected");
63 }
64);
65
66 // Add the controls to a VBox.
67 VBox vbox = new VBox(10, listView, outputLabel, totalButton);
68 vbox.setPadding(new Insets(10));
69 vbox.setAlignment(Pos.CENTER);
70
71 // Create a Scene and display it.
72 Scene scene = new Scene(vbox);
73 primaryStage.setScene(scene);
74 primaryStage.show();
75 }
76 }
```

[Code Listing 12-22](#) demonstrates how you can store `ImageView` controls in a `ListView`. As shown in [Figure 12-21](#), this program displays the images of three poker cards in a horizontal `ListView`. When the user clicks one of the card images, the program displays the name of the card in a `Label` control.

## Figure 12-21 The ListViewDemo6.java application



The window as it initially appears



The window after the user has clicked a card image

[Figure 12-21 Full Alternative Text](#)

## Code Listing 12-22

# ( ListViewDemo6 . java )

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ListView;
5 import javafx.scene.control.Label;
6 import javafx.scene.image.Image;
7 import javafx.scene.image.ImageView;
8 import javafx.scene.layout.VBox;
9 import javafx.geometry.Pos;
10 import javafx.geometry.Insets;
11 import javafx.geometry.Orientation;
12
13 public class Test extends Application
14 {
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Constants for the ListView dimensions
25 final double WIDTH = 425.0, HEIGHT = 200.0;
26
27 // Create the Image objects.
28 Image aceSpadesImage = new Image("file:Ace_Spades.png");
29 Image kingSpadesImage = new Image("file:King_Spades.png");
30 Image queenSpadesImage = new Image("file:Queen_Spades.png")
31
32 // Create the ImageView controls.
33 ImageView aceSpadesIV = new ImageView(aceSpadesImage);
34 ImageView kingSpadesIV = new ImageView(kingSpadesImage);
35 ImageView queenSpadesIV = new ImageView(queenSpadesImage);
36
37 // Create a ListView for the ImageView objects.
38 ListView<ImageView> listView = new ListView<>();
39 listView.setPrefSize(WIDTH, HEIGHT);
40 listView.setOrientation(Orientation.HORIZONTAL);
41 listView.getItems().addAll(aceSpadesIV, kingSpadesIV,
42 queenSpadesIV);
43
44 // Array with the names of the cards
```

```

45 String[] cardNames = {"Ace of Spades", "King of Spades",
46 "Queen of Spades"};
47
48 // Create a Label to display the selection.
49 Label outputLabel = new Label("Select a Card");
50
51 // Create an event handler for the ListView control.
52 listView.getSelectionModel().selectedItemProperty().addListener
53 {
54 // Get the index of the selected card.
55 int index = listView.getSelectionModel().getSelectedIndex
56
57 // Display the selected card name in the Label.
58 outputLabel.setText(cardNames[index]);
59 });
60
61 // Add the controls to a VBox.
62 VBox vbox = new VBox(10, listView, outputLabel);
63 vbox.setPadding(new Insets(10));
64 vbox.setAlignment(Pos.CENTER);
65
66 // Create a Scene and display it.
67 Scene scene = new Scene(vbox);
68 primaryStage.setScene(scene);
69 primaryStage.show();
70 }
71 }
```

Let's take a closer look at the program:

- Line 25 declares constants for the size of the `ListView` control.
- Lines 28 through 30 load the card images and create `Image` objects.
- Lines 33 through 35 create `ImageView` controls for the card images.
- Line 38 creates a `ListView` control to display the card images. Notice the `ListView` data type is `ListView<ImageView>`.
- Line 39 sets the size of the `ListView`, and line 40 sets the `ListView`'s orientation to horizontal.
- Lines 41 and 42 add the `ImageView` controls to the `ListView`.

- Lines 45 and 46 declare a `String` array containing the names of the cards. Notice the following parallel relationship exists between the names of the cards in the array, and the `ImageView` controls in the `ListView`:
  - The string “Ace of Spades” is at subscript 0 in the array, and the image of the Ace of Spades is at index 0 in the `ListView`.
  - The string “King of Spades” is at subscript 1 in the array, and the image of the King of Spades is at index 1 in the `ListView`.
  - The string “Queen of Spades” is at subscript 2 in the array, and the image of the Queen of Spades is at index 2 in the `ListView`.
- Line 49 creates a `Label` control to display the name of the selected card.
- Lines 52 through 59 register an event handler with the `ListView` control. This event handler will execute any time the user clicks an item in the `ListView`:
  - Line 55 gets the index of the card image that was selected in the `ListView`.
  - Line 58 displays the corresponding string from the `cardNames` array. This is the name of the card that was selected in the `ListView`.
- Lines 62 through 64 put all of the controls in a `VBox`, and set the `VBox`’s alignment and padding.
- Lines 67 through 69 create a scene, put the scene on the stage, and display the application’s window.



## Checkpoint

1. 12.19 How do you set the size of a `ListView`?

2. 12.20 How do you retrieve the selected item from a `ListView` control?
3. 12.21 What is single selection mode?
4. 12.22 What is multiple interval selection mode?
5. 12.23 How do you set the orientation of a `ListView` control?

# 12.5 ComboBox Controls

## Concept:

A *ComboBox* allows the user to select an item from a drop-down list.

A *ComboBox* presents a list of items from which the user may select. Unlike a *ListView*, a *ComboBox* presents its items in a drop-down list. You create a *ComboBox* control with the *ComboBox* class (which is in the *javafx.scene.control* package). The following code snippet shows an example of creating a *ComboBox* named *nameComboBox*:



### Video Note ComboBox Controls

```
ComboBox<String> nameComboBox = new ComboBox<>();
```

Notice at the beginning of the statement, the notation *<String>* appears immediately after the word *ComboBox*. This specifies that this particular *ComboBox* control can hold only *String* objects. If we try to store any other type of object in this *ComboBox*, an error will occur when we compile the program. Also notice the diamond operator *<>* appears after *ComboBox* at the end of the statement. (You might recall from [Chapter 7](#) that *ArrayList* objects are declared in a similar fashion.)

Once you have created a *ComboBox* control, you are ready to add items to it. The following statement shows how we can add strings to our *nameComboBox* control:

```
nameComboBox.getItems().addAll("Will", "Megan", "Amanda", "Tyler")
```

The *nameComboBox* control created by this code will initially appear as the

button shown on the left in [Figure 12-22](#). The button displays the item that is currently selected. When the user clicks the button, the drop-down list appears as shown in the image in the middle of the figure. When the user selects an item from the list, it will appear on the ComboBox's button, as shown in the image on the right in the figure.

## Figure 12-22 The ComboBox control



[Figure 12-22 Full Alternative Text](#)



### Note:

Just like the ListView control, the ComboBox control keeps its list of items in an ObservableList object. We discussed the ObservableList interface in [Chapter 11](#). The ComboBox class's getItems() method returns the ObservableList that holds the ComboBox's items.

[Table 12-6](#) lists some of the most often used ComboBox methods.

## Table 12-6 Some of the ComboBox methods

| Method                             | Description                                                                                                                                                |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| getValue()                         | Returns the item that is currently selected in the ComboBox.                                                                                               |
| setValue( <i>value</i> )           | Selects <i>value</i> in the ComboBox.                                                                                                                      |
| setVisibleRowCount( <i>count</i> ) | The <i>count</i> argument is an int. Sets the number of rows, or items, to display in the drop-down list.                                                  |
| setEditable( <i>value</i> )        | The <i>value</i> argument is a boolean. If <i>value</i> is true, the ComboBox will be editable. If <i>value</i> is false, the ComboBox will be uneditable. |
| show()                             | Displays the drop-down list of items.                                                                                                                      |
| hide()                             | Hides the drop-down list of items.                                                                                                                         |
| isShowing()                        | Returns true or false to indicate whether the drop-down list is visible.                                                                                   |

## Retrieving the Selected Item

You can use the ComboBox class's `getValue()` method get the item that is currently selected. For example, assume we have a ComboBox control named `comboBox`. The following code gets the item that is currently selected in the control, and assigns it to a String named `selected`:

```
String selected = comboBox.getValue();
```

If no item is selected in the ComboBox, the method will return null. [Code Listing 12-23](#) shows a complete example. It displays a list of countries in a

ComboBox control. The user selects a country, then clicks the Button control. The Button's event handler gets the selected country and displays it in a Label control. [Figure 12-23](#) shows the program running.

## Figure 12-23 The ComboBoxDemo1.java application



[Figure 12-23 Full Alternative Text](#)

## Code Listing 12-23 (ComboBoxDemo1.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ComboBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Button;
7 import javafx.scene.layout.VBox;
8 import javafx.geometry.Pos;
9 import javafx.geometry.Insets;
```

```

10
11 public class ComboBoxDemo1 extends Application
12 {
13 public static void main(String[] args)
14 {
15 // Launch the application.
16 launch(args);
17 }
18
19 @Override
20 public void start(Stage primaryStage)
21 {
22 // Create a ComboBox.
23 ComboBox<String> comboBox = new ComboBox<>();
24 comboBox.getItems().addAll("England", "Scotland",
25 "Ireland", "Wales");
26
27 // Create a Label.
28 Label outputLabel = new Label("Select a Country");
29
30 // Create a Button.
31 Button button = new Button("Get Selection");
32 button.setOnAction(event ->
33 {
34 outputLabel.setText(comboBox.getValue());
35 });
36
37 // Add the controls to a VBox.
38 VBox vbox = new VBox(10, comboBox, outputLabel, button);
39 vbox.setPadding(new Insets(10));
40 vbox.setAlignment(Pos.CENTER);
41
42 // Create a Scene and display it.
43 Scene scene = new Scene(vbox);
44 primaryStage.setScene(scene);
45 primaryStage.show();
46 }
47 }

```

# Responding to ComboBox Item Selection with an Event Handler

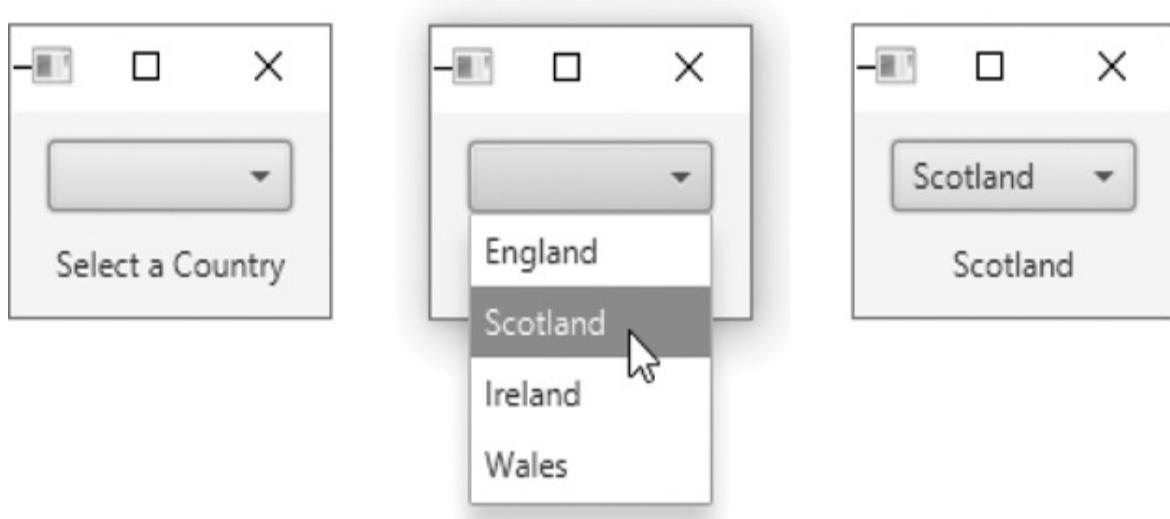
When the user selects an item in a ComboBox, an ActionEvent occurs. If you

want the program to immediately perform an action when the user selects an item in a `ListView`, you can write an event handler that responds to the action event. Assuming `comboBox` is the name of a `ComboBox` control, the following code snippet shows how to use a lambda expression to register such an event handler:

```
comboBox.setOnAction(event ->
{
 // Write event handling code here...
});
```

For example, [Code Listing 12-24](#) is a modification of the program in [Code Listing 12-23](#). This version of the application does not have a `Button` control. Instead, the selected item is displayed immediately after the user makes his or her selection. An example of the program's output is shown in [Figure 12-24](#).

## Figure 12-24 The ComboBoxDemo2.java application



[Figure 12-24 Full Alternative Text](#)

# Code Listing 12-24

## (ComboBoxDemo2.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.ComboBox;
5 import javafx.scene.control.Label;
6 import javafx.scene.layout.VBox;
7 import javafx.geometry.Pos;
8 import javafx.geometry.Insets;
9
10 public class ComboBoxDemo2 extends Application
11 {
12 public static void main(String[] args)
13 {
14 // Launch the application.
15 launch(args);
16 }
17
18 @Override
19 public void start(Stage primaryStage)
20 {
21 // Create a ComboBox.
22 ComboBox<String> comboBox = new ComboBox<>();
23 comboBox.getItems().addAll("England", "Scotland",
24 "Ireland", "Wales");
25
26 // Create a Label.
27 Label outputLabel = new Label("Select a Country");
28
29 // Register an event handler for the ComboBox.
30 comboBox.setOnAction(event ->
31 {
32 outputLabel.setText(comboBox.getValue());
33 });
34
35 // Add the controls to a VBox.
36 VBox vbox = new VBox(10, comboBox, outputLabel);
37 vbox.setPadding(new Insets(10));
38 vbox.setAlignment(Pos.CENTER);
39
40 // Create a Scene and display it.
41 Scene scene = new Scene(vbox);
```

```
42 primaryStage.setScene(scene);
43 primaryStage.show();
44 }
45 }
```

## Editable ComboBoxes

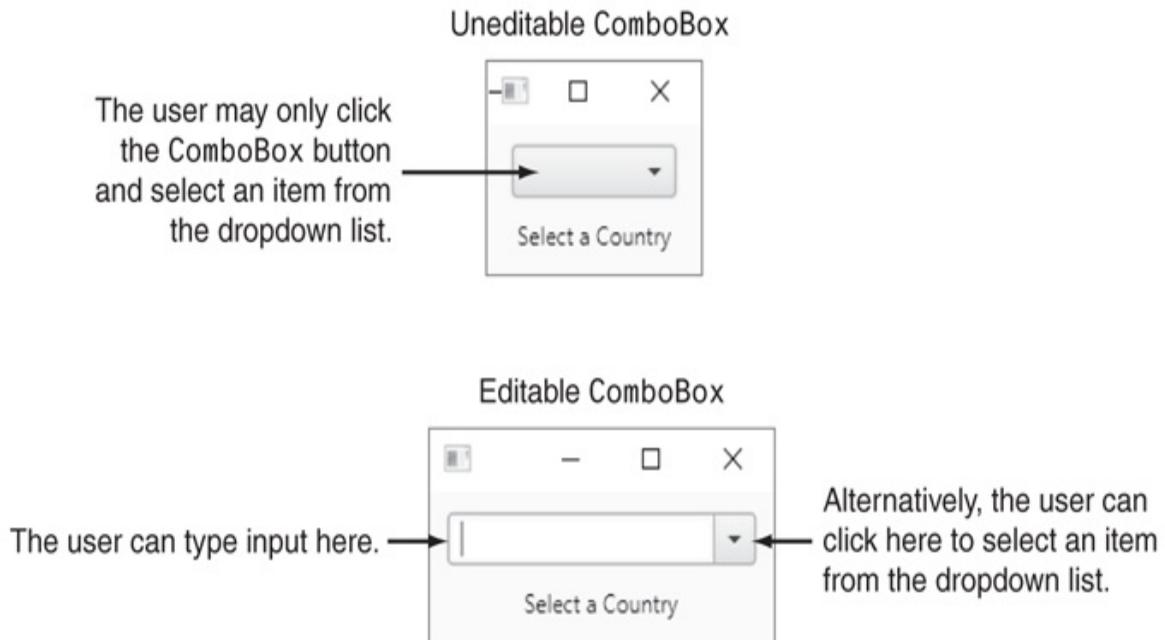
An editable `ComboBox` allows the user to select an item from a drop-down list, or type input into a field that is similar to a `TextField`. By default, `ComboBox` controls are uneditable, which means the user cannot type input into the control; he or she can only select items from a drop-down list.

The `ComboBox` class has a method named `setEditable` that takes a boolean value as its argument. If you pass `true`, the `ComboBox` becomes editable. If you pass `false`, the `ComboBox` becomes uneditable. Here is an example code snippet that demonstrates this:

```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.setEditable(true);
comboBox.getItems().addAll("England", "Scotland",
 "Ireland", "Wales");
```

[Figure 12-25](#) compares an uneditable `ComboBox` with an editable `ComboBox`.

## Figure 12-25 Uneditable versus Editable ComboBoxes



[Figure 12-25 Full Alternative Text](#)



## Checkpoint

1. 12.24 In what type of object does the ComboBox control keep its list of items?
2. 12.25 How do you retrieve the selected item from a ComboBox?
3. 12.26 What type of event do Combox controls generate when they are clicked?
4. 12.27 What is the difference between an editable ComboBox and an uneditable ComboBox?

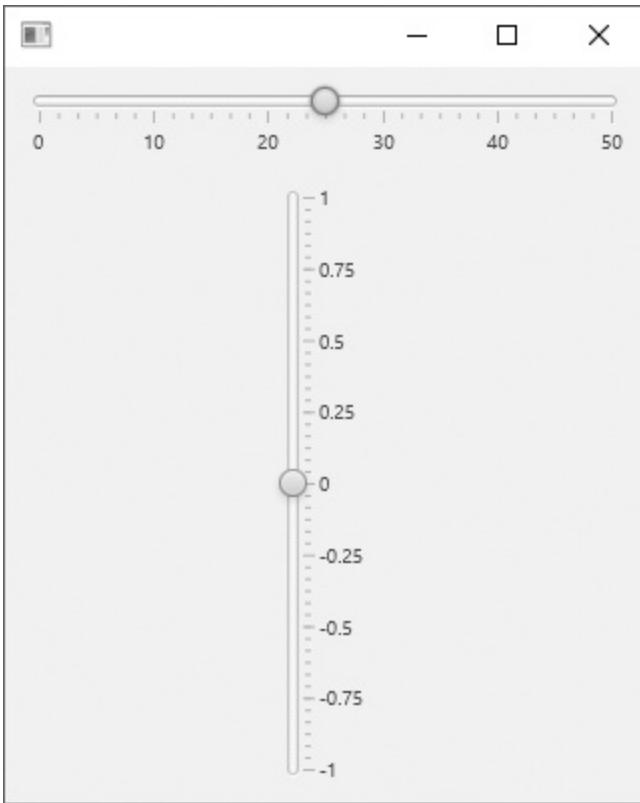
## **12.6 Slider Controls**

### **Concept:**

A **Slider** is a control that allows the user to graphically adjust a number within a range of values.

Slider controls display an image of a “slider knob” that can be dragged along a track. Sliders can be horizontally or vertically oriented, as shown in [Figure 12-26](#). You create Slider controls with the `Slider` class, which is in the `javafx.scene.control` package.

### **Figure 12-26 A horizontal and a vertical Slider**



[Figure 12-26 Full Alternative Text](#)



#### VideoNote Slider Controls

A **Slider** is designed to represent a range of numeric values. At one end of the **Slider** is the range's minimum value, and at the other end is the range's maximum value. In [Figure 12-26](#), the horizontal **Slider**'s minimum value is 0, and its maximum value is 50. The vertical **Slider**'s minimum value is -1, and its maximum value is 1.

**Slider** controls have a numeric value, and as the user moves the knob along the track, the numeric value is adjusted accordingly. Notice the **Slider** controls in [Figure 12-26](#) have accompanying tick marks. Starting at 0, and then at every 10th value, a major tick mark is displayed along with a label indicating the value at that tick mark. Between the major tick marks are minor tick marks. In this example, there are five minor tick marks displayed between the major tick marks.

The `Slider` class has two constructors. The no-arg constructor creates a `Slider` with a minimum value of 0, and a maximum value of 100. The `Slider`'s initial value will be set to 0, and no tick marks will be displayed. Here is a statement that creates a `Slider` using the no-arg constructor:

```
Slider slider = new Slider();
```

The second constructor accepts three `double` arguments: the minimum value, the maximum value, and the initial value. The following statement creates a `Slider` with the minimum value 0, the maximum value 50, and the initial value 25:

```
Slider slider = new Slider(0.0, 50.0, 25.0);
```

The appearance of tick marks, their spacing, and the appearance of labels can be controlled through methods in the `Slider` class. [Table 12-7](#) lists some of the methods you will use most often.

## Table 12-7 Some of the `Slider` class methods

| Method                                   | Description                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setOrientation(orientation)</code> | Sets the <code>Slider</code> 's orientation. The argument should be one of the following enum constants: <code>Orientation.HORIZONTAL</code> or <code>Orientation.VERTICAL</code> . (The <code>Orientation</code> enum is in the <code>javafx.geometry</code> package. By default, <code>Slider</code> controls are horizontal.) |
| <code>setShowTickMarks(value)</code>     | The <code>value</code> argument is a <code>boolean</code> value. If you pass <code>true</code> , the <code>Slider</code> will display tick marks. If you pass <code>false</code> , no tick marks                                                                                                                                 |

`setMajorTickUnit(value)`

will be displayed. (By default, no tick marks are displayed.)

The *value* argument is a double. Sets the major tick mark spacing. (The default value is 25.)

`setMinorTickCount(value)`

The *value* argument is an int. Sets the number of minor tick marks to place between the major tick marks. (The default value is 3.)

`setMax(value)`

The *value* argument is a double. Sets the Slider's maximum value. (By default, the maximum value is 100.)

`setMin(value)`

The *value* argument is a double. Sets the Slider's minimum value. (By default, the minimum value is 0.)

`setSnapToTicks(value)`

The *value* argument is a boolean value. If you pass `true`, the Slider knob will always snap to the nearest tick mark when it is moved. If you pass `false`, the Slider knob will not snap to tick marks. (By default, the Slider knob does not snap.)

`setShowTickLabels(value)`

The *value* argument is a boolean value. If you pass `true`, numeric labels are displayed at the major tick marks. If you pass `false`, no labels will be displayed. (By default, no labels are displayed.)

`setBlockIncrement(value)`

The *value* argument is a double. Sets the amount by which the Slider's knob is moved when the user clicks the Slider's track, or

|                          |                                                                            |
|--------------------------|----------------------------------------------------------------------------|
|                          | uses the keyboard to move the knob. (The default value is 10.)             |
| setValue( <i>value</i> ) | The <i>value</i> argument is a double.<br>Sets the Slider's current value. |
| getValue()               | Returns the Slider's current value, as a double.                           |

The following code snippet creates the horizontal slider that is shown in [Figure 12-26](#):

```

1 Slider hSlider = new Slider(0.0, 50.0, 25.0);
2 hSlider.setShowTickMarks(true);
3 hSlider.setMajorTickUnit(10);
4 hSlider.setMinorTickCount(5);
5 hSlider.setShowTickLabels(true);
6 hSlider.setPrefWidth(300.0);

```

Let's take a closer look at the code:

- Line 1 creates a `Slider` with a minimum value of 0, and a maximum value of 50. The slider's initial value is 25. By default, the `Slider`'s orientation is horizontal.
- Line 2 causes the `Slider` to display tick marks.
- Line 3 sets the major tick mark spacing at 10. As a result, major tick marks will be displayed at the values 0, 10, 20, 30, 40, and 50.
- Line 4 causes 5 minor tick marks to be displayed between the major tick marks.
- Line 5 displays numeric labels at the major tick marks.
- Line 6 sets the width of the `Slider` to 300 pixels.

The following code snippet creates the vertical `Slider` shown in [Figure 12-26](#):

```
1 Slider vSlider = new Slider(-1.0, 1.0, 0.0);
```

```
2 vSlider.setOrientation(Orientation.VERTICAL);
3 vSlider.setShowTickMarks(true);
4 vSlider.setMajorTickUnit(0.25);
5 vSlider.setMinorTickCount(5);
6 vSlider.setShowTickLabels(true);
7 vSlider.setPrefHeight(300.0);
```

Let's take a closer look at the code:

- Line 1 creates a `Slider` with a minimum value of -1, and a maximum value of 1. The `Slider`'s initial value is 0.
- Line 2 sets the `Slider`'s orientation to vertical.
- Line 3 causes the `Slider` to display tick marks.
- Line 4 sets the major tick mark spacing at 0.25. As a result, major tick marks will be displayed at the values -1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, and 1.
- Line 5 causes 5 minor tick marks to be displayed between the major tick marks.
- Line 6 displays numeric labels at the major tick marks.
- Line 7 sets the width of the `Slider` to 300 pixels.

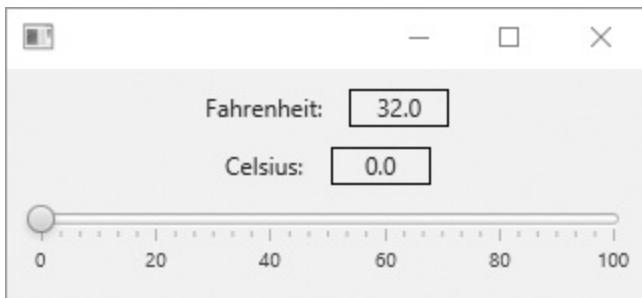
When a `Slider`'s value changes (because the knob was moved), the `Slider` control generates a *change event*. If you want an action to take place when the `Slider`'s value changes, you can register an event handler for the change event. Assuming `slider` is the name of a `Slider` control, the following code shows an example of how to register such an event handler with a lambda expression:

```
slider.valueProperty().addListener(
 (observable, oldvalue, newvalue) ->
{
 // Write event handling code here...
});
```

[Code Listing 12-25](#) demonstrates the `Slider` control. This program displays

the window shown in [Figure 12-27](#). Two temperatures are initially shown: 32.0 degrees Fahrenheit and 0.0 degrees Celsius. A Slider, which has the range of 0 through 100, allows you to adjust the Celsius temperature and immediately see the Fahrenheit conversion.

## Figure 12-27 The SliderDemo.java application



[Figure 12-27 Full Alternative Text](#)

## Code Listing 12-25 (SliderDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.VBox;
5 import javafx.scene.layout.HBox;
6 import javafx.scene.control.Label;
7 import javafx.scene.control.Slider;
8 import javafx.geometry.Pos;
9 import javafx.geometry.Insets;
10
11 public class SliderDemo extends Application
12 {
13 public static void main(String[] args)
14 {
15 // Launch the application.
16 launch(args);
```

```
17 }
18
19 @Override
20 public void start(Stage primaryStage)
21 {
22 // Slider constants
23 final double MIN = 0.0, MAX = 100.0, INITIAL = 0.0;
24 final double MAJOR_TICK_UNIT = 20.0;
25 final int MINOR_TICK_COUNT = 5;
26 final double SLIDER_WIDTH = 300.0;
27
28 // Miscellaneous constants
29 final double LABEL_WIDTH = 50.0;
30 final double SPACING = 10.0;
31
32 // Create the Fahrenheit controls.
33 Label fDescriptor = new Label("Fahrenheit: ");
34 Label fLabel = new Label("32.0");
35 fLabel.setStyle("-fx-border-style: solid;" +
36 "-fx-alignment: center");
37 fLabel.setPrefWidth(LABEL_WIDTH);
38 HBox fHBox = new HBox(SPACING, fDescriptor, fLabel);
39 fHBox.setAlignment(Pos.CENTER);
40
41 // Create the Celsius controls.
42 Label cDescriptor = new Label("Celsius: ");
43 Label cLabel = new Label("0.0");
44 cLabel.setStyle("-fx-border-style: solid;" +
45 "-fx-alignment: center");
46 cLabel.setPrefWidth(LABEL_WIDTH);
47 HBox cHBox = new HBox(SPACING, cDescriptor, cLabel);
48 cHBox.setAlignment(Pos.CENTER);
49
50 // Make the Slider.
51 Slider slider = new Slider(MIN, MAX, INITIAL);
52 slider.setShowTickMarks(true);
53 slider.setMajorTickUnit(MAJOR_TICK_UNIT);
54 slider.setMinorTickCount(MINOR_TICK_COUNT);
55 slider.setShowTickLabels(true);
56 slider.setSnapToTicks(true);
57 slider.setPrefWidth(SLIDER_WIDTH);
58
59 // Register an event handler for the Slider.
60 slider.valueProperty().addListener(
61 (observable, oldValue, newValue) ->
62 {
63 // Get the Celsius temp from the Slider.
```

```

64 double celsius = slider.getValue();
65
66 // Convert Celsius to Fahrenheit.
67 double fahrenheit = (9.0 / 5.0) * celsius + 32;
68
69 // Display the Celsius and Fahrenheit temps.
70 cLabel.setText(String.format("%.1f", celsius));
71 fLabel.setText(String.format("%.1f", fahrenheit));
72 });
73
74 // Add the controls to an VBox.
75 VBox vbox = new VBox(10, fHBox, cHBox, slider);
76 vbox.setAlignment(Pos.CENTER);
77 vbox.setPadding(new Insets(SPACING));
78
79 // Create a Scene and display it.
80 Scene scene = new Scene(vbox);
81 primaryStage.setScene(scene);
82 primaryStage.show();
83 }
84 }
```



## Checkpoint

1. 12.28 What `Slider` methods do you use to perform each of these operations?
  1. Establish the spacing of major tick marks.
  2. Establish the spacing of minor tick marks.
  3. Cause tick marks to be displayed.
  4. Cause labels to be displayed.
2. 12.29 What type of event does a `Slider` generate when its slider knob is moved?
3. 12.30 How do you get a `Slider`'s current value?

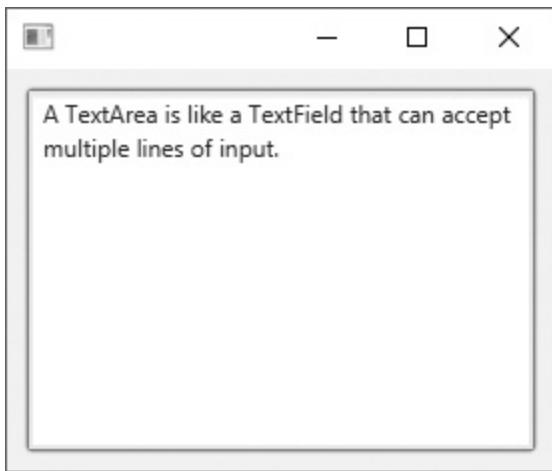
## 12.7 TextArea Controls

### Concept:

A `TextArea` is a multiline `TextField` that can accept several lines of text input, or display several lines of text.

In [Chapter 11](#), you were introduced to the `TextField` control, which allows the user to enter a single line of text. A `TextArea` is like a `TextField` that can accept multiple lines of input. [Figure 12-28](#) shows an example.

### Figure 12-28 The TextArea control



The `TextArea` class, which is in the `javafx.scene.control` package, has two constructors. The no-arg constructor creates an empty `TextArea`. Here is a statement that creates a `TextArea` using the no-arg constructor:

```
TextArea textArea = new TextArea();
```

The second constructor accepts a `String` argument, as shown in this example:

```
TextArea textArea = new TextArea("This is the initial text.");
```

The string that is passed to the `TextArea` constructor will be initially displayed in the `TextArea`. [Table 12-8](#) lists some of the methods you will use most often.

## Table 12-8 Some of the `TextArea` class methods

| Method                                 | Description                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setPrefColumnCount(value)</code> | The <code>value</code> argument is an <code>int</code> . Sets the preferred number of text columns. (The default value is 40.)                                                                                                                                                                                                                                         |
| <code>setPrefRowCount(value)</code>    | The <code>value</code> argument is an <code>int</code> . Sets the preferred number of text rows. (The default value is 10.)                                                                                                                                                                                                                                            |
| <code>setWrapText(value)</code>        | The <code>value</code> argument is a <code>boolean</code> . If the argument is <code>true</code> , then the <code>TextArea</code> will wrap a line of text that exceeds the end of a row. If the argument is <code>false</code> , then the <code>TextArea</code> will scroll when a line of text exceeds the end of a row. (The default value is <code>false</code> .) |
| <code>getText()</code>                 | Returns the contents of the <code>TextArea</code> as a <code>String</code> .                                                                                                                                                                                                                                                                                           |
| <code>setText(value)</code>            | The <code>value</code> argument is a <code>String</code> . The argument becomes the text that is displayed in the <code>TextArea</code> control.                                                                                                                                                                                                                       |

The following code shows an example of creating an empty `TextArea` control with a width of 20 columns and a height of 30 rows:

```
TextArea textArea = new TextArea();
textArea.setPrefColumnCount(20);
textArea.setPrefRowCount(30);
```

By default, `TextArea` controls do not perform *text wrapping*. This means when text is entered into the control and the end of a line is reached, the text does not wrap around to the next line. Instead, the `TextArea` scrolls to the right, and displays a horizontal scrollbar.

As an alternative, you can enable text wrapping with the `TextArea` class's `setWrapText` method. The method accepts a boolean argument. If you pass `true`, then text wrapping is enabled. If you pass `false`, text wrapping is disabled. Here is an example:

```
textInput.setWrapText(true);
```

With line wrapping enabled, a line of text that exceeds the end of a row will be wrapped to the next line.



## Note:

The `TextArea` control performs a type of wrapping known as *word wrapping*. This means the line breaks always occur between words, not in the middle of a word.

To retrieve the text the user has typed into a `TextArea` control, you call the control's `getText` method. The method returns the text that has been entered into the `TextField` as a `String`. For example, assume `textArea` is a `TextArea` control. The following code reads the text that has been entered into `textArea`, and assigns it to a `String` variable named `input`:

```
String input;
input = textArea.getText();
```



## Checkpoint

1. 12.31 What is the difference between a `TextArea` and a `TextField`?
2. 12.32 What is the `TextArea`'s default number of columns displayed? How do you change the number of columns?
3. 12.33 What is the `TextArea`'s default number of rows displayed? How do you change the number of rows?
4. 12.34 What is text wrapping? What is line wrapping? How do you enable text wrapping in a `TextArea` control?
5. 12.35 How do you retrieve the text that the user has typed into a `TextArea` control?

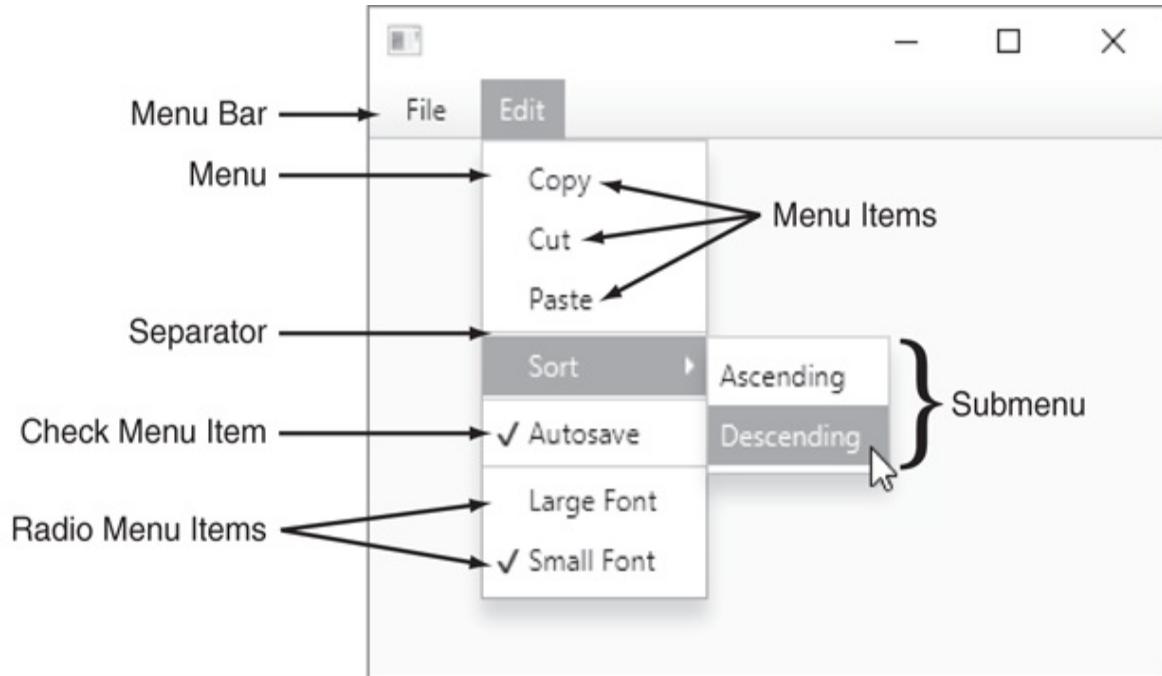
## **12.8 Menus**

### **Concept:**

JavaFX provides classes for creating systems of drop-down menus. Menus can contain menu items, checked menu items, radio button menu items, and other menus.

In the GUI applications you have studied so far, the user initiates actions by clicking on controls such as Buttons. When an application has several operations for the user to choose from, a menu system is more commonly used than buttons. A *menu system* is a collection of commands organized in one or more drop-down menus. Before learning how to construct a menu system, you must learn about the basic items found in a typical menu system. Look at the example menu system in [Figure 12-29](#).

### **Figure 12-29 Example menu system**



[Figure 12-29 Full Alternative Text](#)

The menu system in the figure consists of the following items:

- **Menu bar.** At the top of the window is a menu bar. The *menu bar* lists the names of one or more menus. The menu bar in [Figure 12-29](#) shows the names of two menus: *File* and *Edit*.
- **Menu.** A *menu* is a drop-down list of menu items. The user may activate a menu by clicking on its name on the menu bar. In [Figure 12-29](#), the *Edit* menu has been activated.
- **Menu Item.** A *menu item* can be selected by the user. When a menu item is selected, some type of action is usually performed.
- **Check menu item.** A *check menu item* appears that may be selected or deselected. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. Check menu items are normally used to turn an option on or off. The user toggles the state of a check menu item each time he or she selects it.
- **Radio menu item.** A *radio menu item* may be selected or deselected. A

radio menu item looks like a check menu item. When it is selected, a check mark appears next to the item. When it is deselected, the check does not appear. When a set of radio menu items are grouped with a `ToggleGroup` object, only one of them can be selected at a time. When the user selects a radio menu item, the one that was previously selected is deselected.

- Submenu. A menu within a menu is called a *submenu*. Some of the commands on a menu are actually the names of submenus. You can tell when a command is the name of a submenu because a small right arrow appears to its right. Activating the name of a submenu causes the submenu to appear. For example, in [Figure 12-29](#), clicking on the *Sort* command causes a submenu to appear.
- Separator bar. A separator bar is a horizontal bar used to separate groups of items on a menu. Separator bars are only used as a visual aid and cannot be selected by the user.

A menu system is constructed with the following classes:

**MenuBar.** Use this class to create a menu bar. A `MenuBar` object can contain `Menu` objects.

**Menu.** Use this class to create a menu that drops down from the menu bar. A `Menu` object can contain `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` objects, as well as other `Menu` objects. A submenu is a `Menu` object that is inside another `Menu` object.

**MenuItem.** Use this class to create a regular menu item. A `MenuItem` object generates an action event when the user selects it.

**CheckMenuItem.** Use this class to create a check menu item. The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise. A `CheckMenuItem` object generates an action event when the user selects it.

**RadioMenuItem.** Use this class to create a radio button menu item. `RadioMenuItem` objects can be grouped together in a `ToggleGroup` object so only one of them can be selected at a time. The class's `isSelected` method

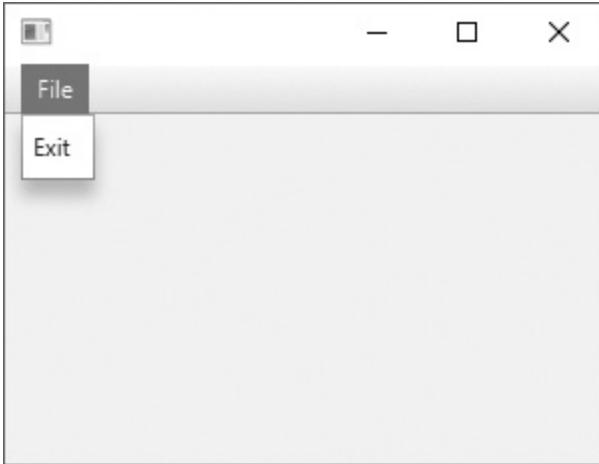
returns true if the item is selected, or false otherwise. A `RadioMenuItem` object generates an action event when the user selects it.

All of these classes are in the `javafx.scene.control` package. A menu system is a `MenuBar` that contains one or more `Menu` objects. Each `Menu` object can contain `MenuItem`, `RadioMenuItem`, and `CheckMenuItem` objects, as well as other `Menu` objects. The process of building a menu system requires the following steps (although not necessarily in this order):

- Create a `MenuBar` object. This will be the “root” of the menu system.
- Create the necessary `Menu` objects, and add them to the `MenuBar` object.
- For each `Menu` object, create the necessary menu item objects, and add them to their `Menu` object.
- Register an event handler for each menu item object.
- Add the `MenuBar` object to the scene graph.

To see an example of an application that uses a simple menu system, look at [Code Listing 12-26](#). The program creates a menu bar with one menu: *File*. The *File* menu has one menu item: *Exit*. When the user selects the *Exit* menu item, the program stops. [Figure 12-30](#) shows the program’s menu system.

## Figure 12-30 The SimpleMenu.java application



## Code Listing 12-26 (**SimpleMenu.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.MenuBar;
5 import javafx.scene.control.Menu;
6 import javafx.scene.control.MenuItem;
7 import javafx.scene.layout.BorderPane;
8
9 public class SimpleMenu extends Application
10 {
11 public static void main(String[] args)
12 {
13 // Launch the application.
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Constants for the scene dimensions
21 final double WIDTH = 300.0, HEIGHT = 200.0;
22
23 // Create the menu bar.
24 MenuBar menuBar = new MenuBar();
25
26 // Create the File menu.
27 Menu fileMenu = new Menu("File");
```

```

28 MenuItem exitItem = new MenuItem("Exit");
29 fileMenu.getItems().add(exitItem);
30
31 // Register an event handler for the exit item.
32 exitItem.setOnAction(event ->
33 {
34 primaryStage.close();
35 });
36
37 // Add the File menu to the menu bar.
38 menuBar.getMenus().add(fileMenu);
39
40 // Add the menu bar to a BorderPane.
41 BorderPane borderPane = new BorderPane();
42 borderPane.setTop(menuBar);
43
44 // Create a Scene and display it.
45 Scene scene = new Scene(borderPane, WIDTH, HEIGHT);
46 primaryStage.setScene(scene);
47 primaryStage.show();
48 }
49 }
```

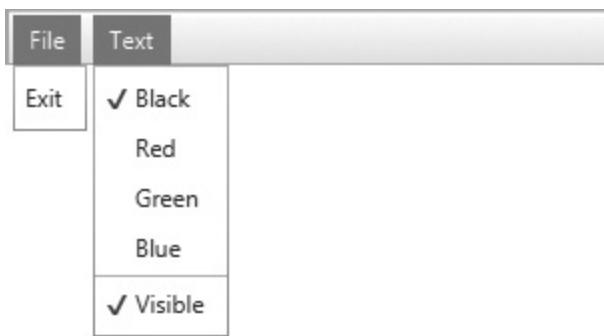
Let's take a closer look at the program:

- Line 21 declares constants for the scene dimensions.
- Line 24 creates a `MenuBar` object.
- Line 27 creates a `Menu` object for the *File* menu.
- Line 28 creates a `MenuItem` object for the *Exit* menu item.
- Line 29 adds the *Exit* menu item to the *File* menu.
- Lines 32 through 35 register an event handler for the *Exit* menu item. When the user clicks the *Exit* menu item, line 34 closes the program.
- Line 38 adds the *File* menu to the menu bar.
- Lines 41 and 42 create a `BorderPane` layout container and add the menu bar to the top region.

- Lines 45 through 47 create the scene, set the scene to the stage, and display the application’s window.

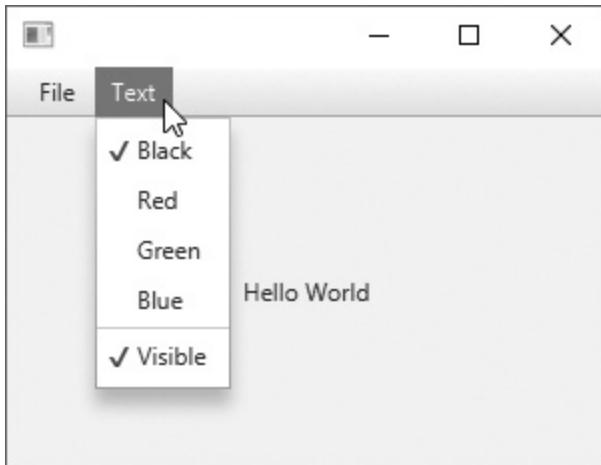
[Code Listing 12-27](#) is an application with a slightly more complex menu system, which is shown in [Figure 12-31](#). This application demonstrates how the text displayed by a `Label` control appears in different colors.

## Figure 12-31 Menu system in the `TextMenu.java` program



The menu bar has two menus: *File* and *Text*. When the user opens the *File* menu, and selects the *Exit* item, the application ends. When the user opens the *Text* menu, he or she can select a color using the *Black*, *Red*, *Green*, and *Blue* radio menu items. These items change the color of the “Hello World” text that is displayed in a `Label` control. The *Text* menu also has a *Visible* item, which is a check menu item. When this item is selected (checked), the `Label` control is made visible. When this item is deselected (unchecked), the `Label` control is made invisible. The application’s window is shown in [Figure 12-32](#).

## Figure 12-32 The `TextMenu.java` application



[Figure 12-32 Full Alternative Text](#)

## Code Listing 12-27 (TextMenu.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Label;
5 import javafx.scene.controlMenuBar;
6 import javafx.scene.control.Menu;
7 import javafx.scene.control.MenuItem;
8 import javafx.scene.control.CheckMenuItem;
9 import javafx.scene.control.RadioButton;
10 import javafx.scene.control.SeparatorMenuItem;
11 import javafx.scene.control.ToggleGroup;
12 import javafx.scene.layout.BorderPane;
13
14 public class TextMenu extends Application
15 {
16 // Fields for the menu components
17 privateMenuBar;
18 private Menu fileMenu;
19 private Menu textMenu;
20 private MenuItem exitItem;
21 private RadioButton blackItem;
22 private RadioButton redItem;
23 private RadioButton greenItem;
24 private RadioButton blueItem;
25 private CheckMenuItem visibleItem;
26
27 // Field for the text
```

```
28 private Label outputLabel;
29
30 public static void main(String[] args)
31 {
32 // Launch the application.
33 launch(args);
34 }
35
36 @Override
37 public void start(Stage primaryStage)
38 {
39 // Constants for the scene dimensions
40 final double WIDTH = 300.0, HEIGHT = 200.0;
41
42 // Create the Label control.
43 outputLabel = new Label("Hello World");
44
45 // Create the menu bar.
46 menuBar = newMenuBar();
47
48 // Create the File menu.
49 buildFileMenu(primaryStage);
50
51 // Create the Text menu.
52 buildTextMenu();
53
54 // Add the File & Text menus to the menu bar.
55 menuBar.getMenus().add(fileMenu);
56 menuBar.getMenus().add(textMenu);
57
58 // Add the controls to a BorderPane.
59 BorderPane borderPane = new BorderPane();
60 borderPane.setTop(menuBar);
61 borderPane.setCenter(outputLabel);
62
63 // Create a Scene and display it.
64 Scene scene = new Scene(borderPane, WIDTH, HEIGHT);
65 primaryStage.setScene(scene);
66 primaryStage.show();
67 }
68
69 /**
70 * This method builds the File menu.
71 */
72
73 private void buildFileMenu(Stage primaryStage)
74 {
75 // Create the File Menu object.
```

```
76 fileMenu = new Menu("File");
77
78 // Create the Exit MenuItem object.
79 exitItem = new MenuItem("Exit");
80
81 // Register an event handler for the Exit item.
82 exitItem.setOnAction(event ->
83 {
84 primaryStage.close();
85 });
86
87 // Add the Exit item to the File menu.
88 fileMenu.getItems().add(exitItem);
89 }
90
91 /**
92 * This method builds the Text menu.
93 */
94
95 private void buildTextMenu()
96 {
97 // Create the Text Menu object.
98 textMenu = new Menu("Text");
99
100 // Create the menu items for the Text menu.
101 blackItem = new RadioMenuItem("Black");
102 redItem = new RadioMenuItem("Red");
103 greenItem = new RadioMenuItem("Green");
104 blueItem = new RadioMenuItem("Blue");
105 visibleItem = new CheckMenuItem("Visible");
106
107 // Select the Black and Visible menu items.
108 blackItem.setSelected(true);
109 visibleItem.setSelected(true);
110
111 // Add the RadioMenuItems to a ToggleGroup.
112 ToggleGroup textToggleGroup = new ToggleGroup();
113 blackItem.setToggleGroup(textToggleGroup);
114 redItem.setToggleGroup(textToggleGroup);
115 greenItem.setToggleGroup(textToggleGroup);
116 blueItem.setToggleGroup(textToggleGroup);
117
118 // Register event handlers for the menu items.
119 blackItem.setOnAction(event ->
120 {
121 outputLabel.setStyle("-fx-text-fill: black");
122 });

```

```

123
124 redItem.setOnAction(event ->
125 {
126 outputLabel.setStyle("-fx-text-fill: red");
127 });
128
129 greenItem.setOnAction(event ->
130 {
131 outputLabel.setStyle("-fx-text-fill: green");
132 });
133
134 blueItem.setOnAction(event ->
135 {
136 outputLabel.setStyle("-fx-text-fill: blue");
137 });
138
139 visibleItem.setOnAction(event ->
140 {
141 if (outputLabel.isVisible())
142 outputLabel.setVisible(false);
143 else
144 outputLabel.setVisible(true);
145 });
146
147 // Add the menu items to the Text menu.
148 textMenu.getItems().add(blackItem);
149 textMenu.getItems().add(redItem);
150 textMenu.getItems().add(greenItem);
151 textMenu.getItems().add(blueItem);
152 textMenu.getItems().add(new SeparatorMenuItem());
153 textMenu.getItems().add(visibleItem);
154 }
155 }
```

Lines 17 through 25 declare private fields for the menu components. We declared these fields because the code that builds the menu system is modularized into methods, and we need to access these objects from the different methods. Line 28 declares `outputLabel` as a private field for the same reason.

Let's take a closer look at the `start` method, in lines 37 through 67:

- Line 40 declares constants for the scene dimensions.
- Line 43 creates the `outputLabel` control.

- Line 46 creates the `MenuBar`.
- Line 49 calls the `buildFileMenu` method, which builds the *File* menu. Notice we are passing the `primaryStage` variable as an argument. Recall that the `primaryStage` variable references the application's `Stage`. The `buildFileMenu` method needs to access the `Stage` because the statement in line 84 calls the `Stage` object's `close()` method to end the application.
- Line 52 calls the `buildTextMenu` method, which builds the *Text* menu.
- Lines 55 and 56 add the *File* menu and the *Text* menu to the menu bar.
- Lines 59 through 61 create a `BorderPane`, put the menu bar in the top region, and place the `Label` control in the center region.
- Lines 64 through 66 create the scene, set the scene to the stage, and display the application's window.

The `buildFileMenu` method appears in lines 73 through 89. When this method is finished, the *File* menu will be constructed, and an event handler will be registered for that menu's *Exit* item.

The `buildTextMenu` method appears in lines 95 through 154. When this method is finished, the *Text* menu will be constructed, and event handlers will be registered for all of that menu's items.

## Assigning Mnemonics to Menu Items

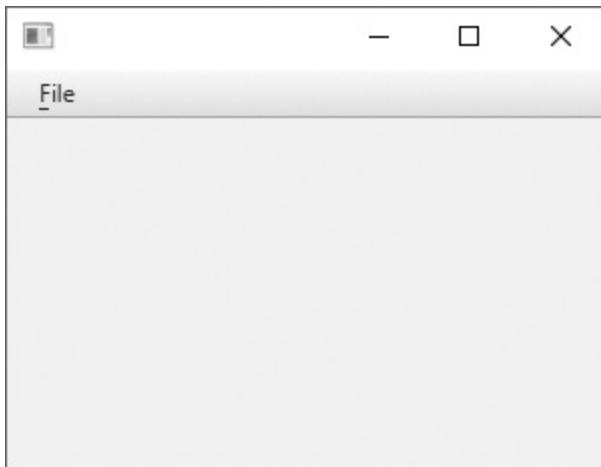
A *mnemonic* is a key on the keyboard that you press in combination with the Alt key to access a menu item quickly. Mnemonics are sometimes referred to as *shortcut keys*, *access keys*, or *hot keys*. You assign a mnemonic to a menu item when you instantiate the menu item. The following code snippet shows an example:

```
Menu fileMenu = new Menu("_File");
MenuItem exitItem = new MenuItem("E_xit");
```

Notice the string we are passing to the `Menu` class constructor in the first statement. An underscore character appears immediately before the letter 'F'. This will cause the letter F to become a mnemonic. When the user presses Alt+F on the keyboard, the *File* menu will be automatically selected. Also notice the string that we are passing to the `MenuItem` class constructor in the second statement. An underscore character appears immediately before the letter 'x'. This will cause the letter X to become a mnemonic. When the user presses Alt+X on the keyboard, the *Exit* menu item will be selected.

The underscore character will not be displayed in the name of the menu item at runtime. However, when the user presses the Alt key on the keyboard, the mnemonic character will appear underlined, as shown in [Figure 12-33](#).

## Figure 12-33 The *File* menu with the Alt+F mnemonic



### Note:

Mnemonics do not distinguish between uppercase and lowercase characters.

There is no difference between Alt+X and Alt+x.



## Checkpoint

1. 12.36 Briefly describe each of the following menu system items:
  1. Menu bar
  2. Menu
  3. Menu item
  4. Check menu item
  5. Radio menu item
  6. Submenu
  7. Separator bar
2. 12.37 What class do you use to create a menu bar?
3. 12.38 What class do you use to create a menu?
4. 12.39 What class do you use to create a menu item?
5. 12.40 What class do you use to create a radio menu item? How do you cause it to be initially selected?
6. 12.41 How do you create a relationship between radio menu items so only one may be selected at a time?
7. 12.42 What class do you use to create a check menu item? How do you cause it to be initially selected?
8. 12.43 What type of event do menu items generate when selected by the user?

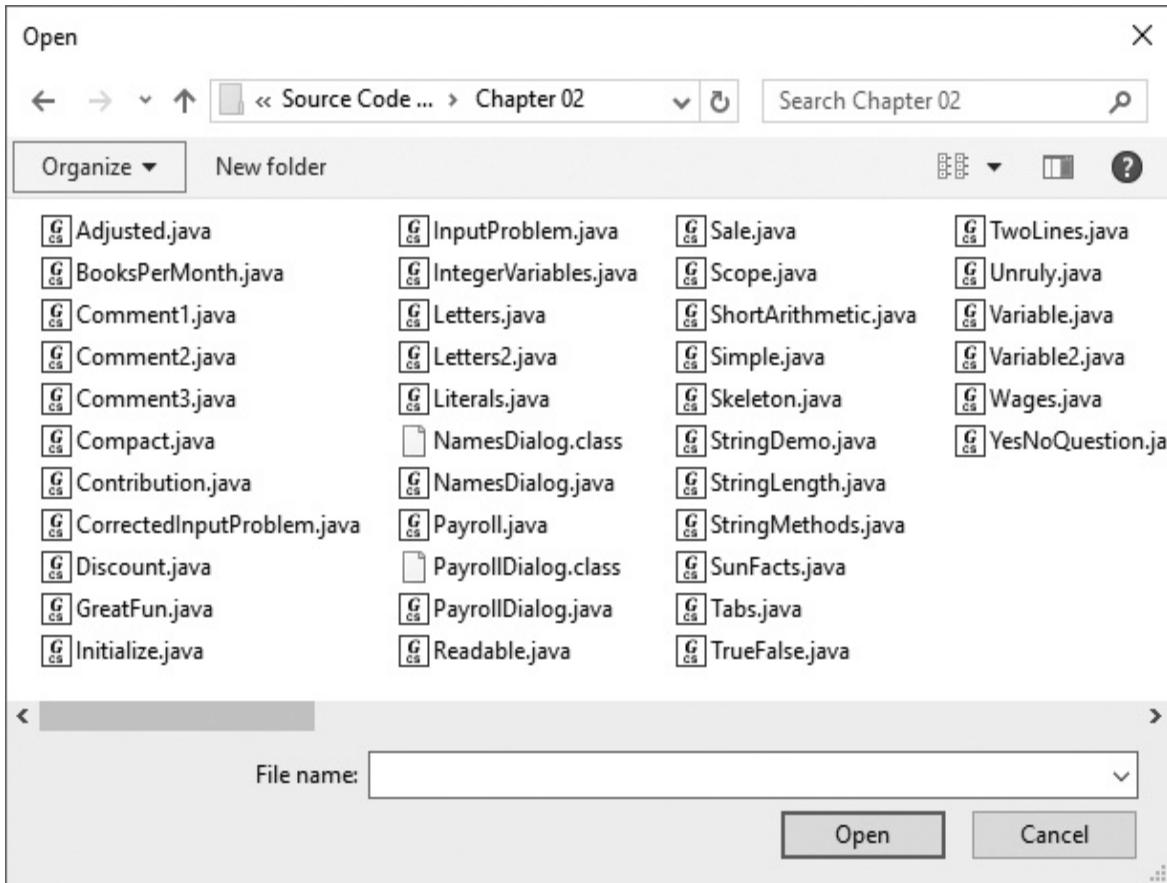
## 12.9 The `FileChooser` Class

### Concept:

*JavaFX provides a `FileChooser` class that equips your applications with standard dialog boxes for opening files and saving files.*

The `FileChooser` class displays a specialized dialog box that allows the user to browse for a file and select it. [Figure 12-34](#) shows an example of a file chooser dialog box on a Windows system.

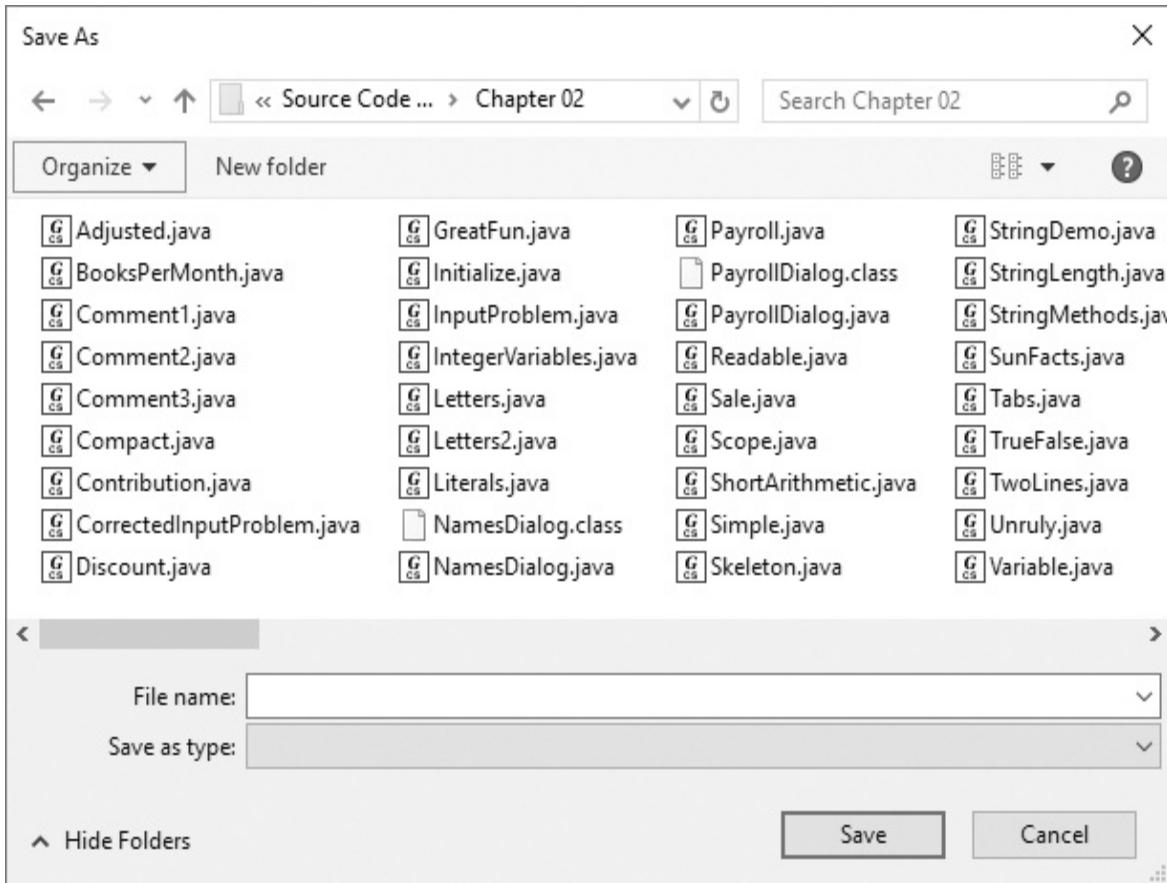
**Figure 12-34 A file chooser dialog box for opening a file**



[Figure 12-34 Full Alternative Text](#)

The `FileChooser` class is in the `javafx.stage` package. The `FileChooser` class can display two types of predefined dialog boxes: an *open dialog box* and a *save dialog box*. The dialog box shown in [Figure 12-34](#) is an open dialog box. This dialog box lets the user browse for an existing file to open. A save dialog box, as shown in [Figure 12-35](#), is employed when the user needs to browse to a location to save a file. Both of these dialog boxes are similar in the way they operate.

## Figure 12-35 A file chooser dialog box for saving a file



[Figure 12-35 Full Alternative Text](#)

## Displaying a FileChooser Dialog Box

To display either an open dialog box or a save dialog box, you first create an instance of the `FileChooser` class. Then, you call either the `showOpenDialog` method (if you want to display an open dialog box), or the `showSaveDialog` method (if you want to display a save dialog). With either method, you pass a reference to the application's stage as an argument. The following code snippet shows an example of calling the `showOpenDialog` method:

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
```

In this code, assume `primaryStage` references our application's stage. The `showOpenDialog` and `showSaveDialog` methods return a `File` object (the `File` class, which was introduced in [Chapter 5](#), is in the `java.io` package) containing information about the selected file. If the user does not select a file, the method returns `null`. You can typically use the `File` object to open the file, or create a file. The following code snippet shows how you can use the `File` object's `getPath` method to get the path and file name as a `String`:

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
if (selectedFile != null)
{
 String filename = selectedFile.getPath();
 outputLabel.setText("You selected " + filename);
}
```



## Checkpoint

1. 12.44 In what package is the `FileChooser` class?
2. 12.45 How do you display an open dialog?
3. 12.46 How do you display a save dialog?
4. 12.47 How do you determine the file that the user selected with either an open dialog or a save dialog?

# 12.10 Using Console Output to Debug a GUI Application

## Concept:

*When debugging a GUI application, you can use `System.out.print`, `System.out.println`, or `System.out.printf` to send diagnostic messages to the console.*

When an application is not performing correctly, programmers sometimes write statements that display *diagnostic messages* into the application. For example, if an application is not giving the correct result for a calculation, diagnostic messages can be displayed at various points in the program’s execution showing the values of all the variables used in the calculation. If the trouble is caused by a variable that has not been properly initialized, or that has not been assigned the correct value, the diagnostic messages reveal this problem. This helps the programmer to see what is going on “under the hood” while an application is running.

The console output methods, `System.out.print`, `System.out.println` and `System.out.printf`, can be valuable tools for displaying diagnostic messages in a GUI application. Because these methods send their output to the console, diagnostic messages can be displayed without interfering with the application’s GUI windows. [Code Listing 12-28](#) shows an example. This is a modified version of the LambdaKiloConverter program that you saw in [Chapter 11](#). In this version of the program, we have inserted statements that write console messages inside the `calcButton`’s event handler. The lines containing the calls are shown in bold. These calls display the value that the application has retrieved from the text field, and is working within its calculation.

# Code Listing 12-28

## (ConsoleDebugging.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.VBox;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import javafx.scene.control.Label;
9 import javafx.scene.control.TextField;
10 import javafx.scene.control.Button;
11 import javafx.event.EventHandler;
12 import javafx.event.ActionEvent;
13
14 /**
15 * Kilometer Converter application with console debugging
16 */
17
18 public class ConsoleDebugging extends Application
19 {
20 // Fields
21 private TextField kiloTextField;
22 private Label resultLabel;
23
24 public static void main(String[] args)
25 {
26 // Launch the application.
27 launch(args);
28 }
29
30 @Override
31 public void start(Stage primaryStage)
32 {
33 // Create a Label to display a prompt.
34 Label promptLabel = new Label("Enter a distance in kilomete
35
36 // Create a TextField for input.
37 kiloTextField = new TextField();
38
39 // Create a Button to perform the conversion.
40 Button calcButton = new Button("Convert");
41
42 // Set up the scene.
43 Scene scene = new Scene(new HBox(promptLabel, kiloTextField, calcButton), 300, 200);
44 primaryStage.setScene(scene);
45 primaryStage.show();
46 }
47 }
```

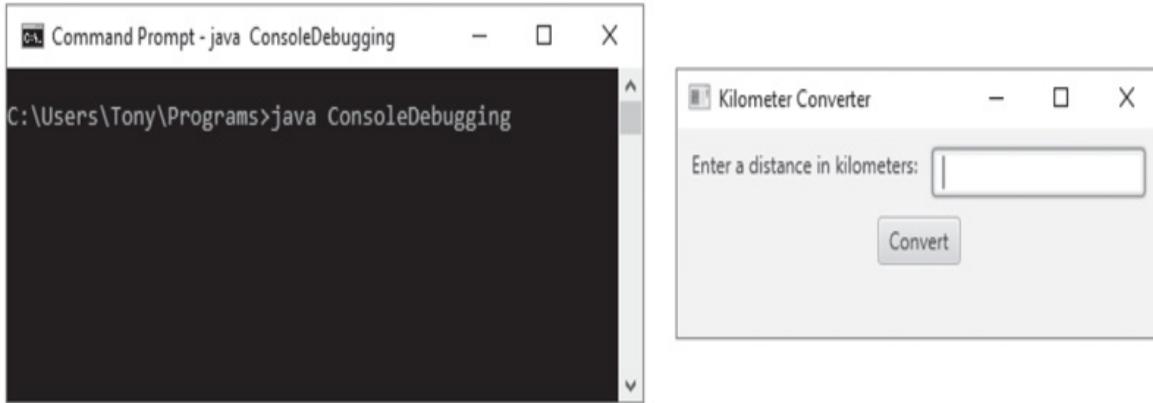
```
42 // Create an event handler.
43 calcButton.setOnAction(event ->
44 {
45 // For debugging, display the text entered, and
46 // its value converted to a double.
47 System.out.printf("Reading %s from the text field.\n",
48 kiloTextField.getText());
49 System.out.printf("Converted value: %f\n",
50 Double.parseDouble(kiloTextField.getText()));
51
52 // Get the kilometers.
53 Double kilometers = Double.parseDouble(kiloTextField.getT
54
55 // Convert the kilometers to miles.
56 Double miles = kilometers * 0.6214;
57
58 // Display the results.
59 resultLabel.setText(String.format("%,.2f miles", miles));
60
61 // For debugging, display a message indicating
62 // the application is ready for more input.
63 System.out.println("Ready for the next input.");
64});
65
66 // Create an empty Label to display the result.
67 resultLabel = new Label();
68
69 // Put the promptLabel and the kiloTextField in an HBox.
70 HBox hbox = new HBox(10, promptLabel, kiloTextField);
71
72 // Put the HBox, calcButton, and resultLabel in a VBox.
73 VBox vbox = new VBox(10, hbox, calcButton, resultLabel);
74
75 // Set the VBox's alignment to center.
76 vbox.setAlignment(Pos.CENTER);
77
78 // Set the VBox's padding to 10 pixels.
79 vbox.setPadding(new Insets(10));
80
81 // Create a Scene.
82 Scene scene = new Scene(vbox);
83
84 // Add the Scene to the Stage.
85 primaryStage.setScene(scene);
86
87 // Set the stage title.
88 primaryStage.setTitle("Kilometer Converter");
89
```

```
90 // Show the window.
91 primaryStage.show();
92 }
93 }
```

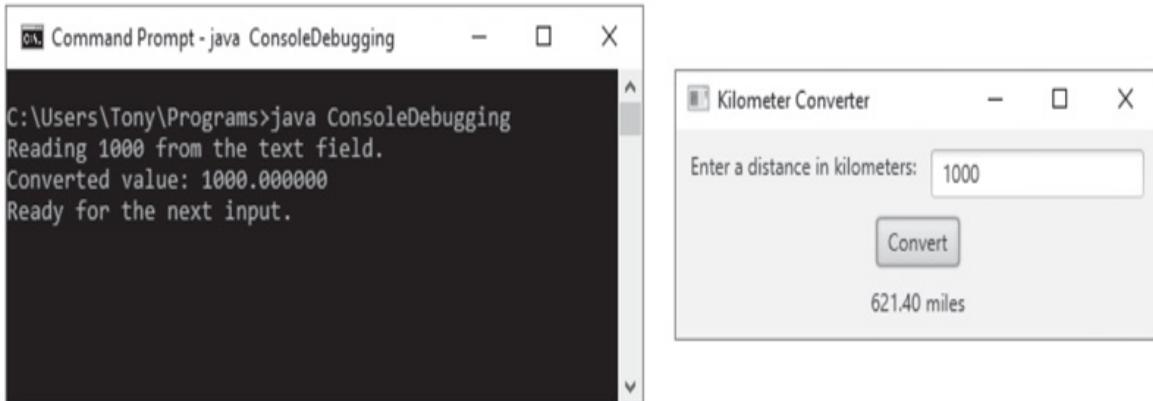
Let's take a closer look. In lines 47 and 48, a message is displayed to the console showing the value that was read from the text field. In lines 49 and 50, another message is displayed showing the value after it is converted to a double. Then, in line 63, a message is displayed indicating the application is ready for its next input. [Figure 12-36](#) shows an example session with the application on a computer running Windows. Both the console window and the application windows are shown.

## **Figure 12-36 Messages displayed to the console during the application's execution**

1. A command is typed in the console window to execute the application. The application's window appears.



2. The user types a value into the text field and clicks the Convert button. Debugging messages appear in the console window.



[Figure 12-36 Full Alternative Text](#)

The messages displayed to the console are meant only for the programmer to see while he or she is debugging the application. Once the programmer is satisfied that the application is running correctly, the statements that display the messages can be taken out.

## 12.11 Common Errors to Avoid

- Forgetting to add `RadioButton` controls to a `ToggleGroup`. A mutually exclusive relationship is created between `RadioButton` controls only when they are added to a `ToggleGroup` object.
- Forgetting to initially select one of the `RadioButton` controls in a `ToggleGroup`. In most situations, you want one of the `RadioButton` controls in a `ToggleGroup` to be initially selected. In code, you can select a `RadioButton` control by calling its `setSelected` method, passing `true` as the argument.
- Adding items to a `ListView` when you really meant to set items, or vice versa. The `ListView` control's `getItems().addAll()` method adds items to the `ListView`. If the `ListView` already contains items, the `getItems().addAll()` method will not erase the existing items, but will add the new items to the existing list. The `getItems().setAll()` method, on the other hand, replaces a `ListView` control's existing items with a new list of items.
- Forgetting to add menu items to a `Menu` control, and a `Menu` control to a `MenuBar` control. After you create a menu item, you must add it to a `Menu` control in order for it to be displayed on the menu. Likewise, `Menu` controls must be added to a `MenuBar` control in order to be displayed on the `MenuBar`.
- Not grouping `RadioMenuItem` controls in a `ToggleGroup` object. Just like regular `RadioButton` controls, you must group `RadioMenuItem` controls in a `ToggleGroup` in order to create a mutually exclusive relationship between them.

# **Review Questions**

## **Multiple Choice and True/False**

1. When a selector name starts with a period in a JavaFX CSS style definition, it means the selector corresponds to
  1. specific variable names in the application
  2. specific types of JavaFX nodes
  3. values that the user enters
  4. named constants in the application
2. Which of these is written in the correct syntax for a CSS style definition?
  1. `{.label: -fx-font-size: 20pt; }`
  2. `.label { -fx-font-size=20pt; }`
  3. `.label { -fx-font-size: 20pt; }`
  4. `{.label; -fx-font-size: 20pt; }`
3. If you want to apply styles to all of the nodes in a scene, use the selector.
  1. `.all-nodes`
  2. `.stage`
  3. `.home`

4. .root
4. The CSS property is used to change a control's background color in a JavaFX application.
  1. -fx-color
  2. -fx-background-color
  3. -fx-color-background
  4. -fx-bg-color
5. When you define a color in CSS, you specify an integer value in the range of for each of the color components (red, green, and blue).
  1. 0 through 255
  2. 0 through 1024
  3. -255 through 255
  4. 0 through 128
6. In the hexadecimal color value #05AAFF, the AA value specifies which color component?
  1. red
  2. green
  3. blue
  4. magenta
7. What style definition applies a background color of #0000FF to all Button controls?

1. .button { -fx-background-color: #0000FF; }
  2. .button { -fx-bg-color = #0000FF; }
  3. .button.all { -fx-background-color: #0000FF; }
  4. .global {button: -fx-background-color: #0000FF; }
8. Adding RadioButton controls to this type of object creates a mutually exclusive relationship between them.
1. MutualExclude
  2. RadioGroup
  3. LogicalGroup
  4. ToggleGroup
9. This RadioButton method returns true if this control is selected.
1. selected()
  2. clicked()
  3. isSelected()
  4. isChecked()
10. The **control** displays a list of items and allows the user to select one or more items from the list.
1. DropDownList
  2. ListView
  3. ViewList
  4. ItemList

11. The control presents its items in a drop-down list.

1. `DropList`
2. `ListView`
3. `ComboBox`
4. `ItemList`

12. Both the `ListView` control and the `ComboBox` control keeps its list of items in a(n) object.

1. `ArrayList`
2. `ExpandableList`
3. `ItemList`
4. `ObservableList`

13. A is like a `TextField` that can accept multiple lines of input.

1. `TextArea`
2. `MultilineTextField`
3. `ExpandableTextField`
4. `TextEditorField`

14. You use this class to create a menu bar.

1. `MenuItem`
2. `MenuBar`

3. Menu

4. Bar

15. You use this class to create a radio menu item.

1. MenuItem

2. RadioButton

3. RadioItem

4. RadioMenuItem

16. True or False: If you make any changes to an application's stylesheet, you have to recompile the Java application in order for the style changes to take effect.

17. True or False: If a style definition for a specific type of node contradicts a style rule in the .root definition, the more specific style definition takes precedence over the .root definition.

18. True or False: You can list only one selector in the first line of a style definition.

19. True or False: A node's `setStyle` method removes any styles that were previously applied to the node, and replaces them with the new style rule that is passed as an argument.

20. True or False: A scroll bar automatically appears when a `ListView` contains more items than can be displayed in the space provided.

21. True or False: By default, `ListView` controls are oriented horizontally.

22. True or False: By default, `TextArea` controls perform line wrapping.

23. True or False: A `MenuBar` object acts as a container for `Menu` objects.

24. True or False: A `Menu` object cannot contain other `Menu` objects.

# Find the Error

1. 

```
.button { -fx-background-color = #0000FF; }
```
2. 

```
.label { -font-size: 14pt; }
```
3. 

```
// This code has an error!
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
ToggleGroup radioGroup = new ToggleGroup();
radioGroup.setToggleGroup(radio1);
radioGroup.setToggleGroup(radio2);
```
4. 

```
// This code has an error!
ListView<String> myListView = new ListView<>();
myListView.getItems().addAll(10, 20, 30, 40);
```

# Algorithm Workbench

1. Write a JavaFX CSS style definition that will be applied to all `Label` controls. The style definition should specify the following: font size: 24, font style: italic, font weight: bold.
2. Suppose we have a stylesheet named `styles.css`, and in our Java code the `scene` variable references the `Scene` object. Write a statement to add the stylesheet to the scene.
3. Write a JavaFX CSS style definition that will be applied to all `Label`, `Button`, and `TextField` controls. The style definition should specify the following: font size: 24, font style: italic, font weight: bold.
4. Assume `scene` is the name of a `Scene` object, and `styles.css` is the name of a CSS stylesheet. Write the statement that applies the `styles.css` stylesheet to `scene`.
5. Write functional RGB notation for a color where red = 100, green = 20, and blue = 255.

6. An application has a `Button` control named `calcButton`, and you have applied a stylesheet to the root node. In that style sheet, you have defined a custom style class named `button-red`. Write the statement that would apply the `button-red` custom style class to the `calcButton` control.
7. An application has a `Button` control named `calcButton`. Write the statement that assigns the ID "calc-button" to the `calcButton` control.
8. An application has a `Label` control named `myLabel`. Write a statement that applies an inline style rule to `myLabel`. The style rule should set the font size to 12 points.
9. Write the code to add the following `RadioButton` controls to the `ToggleGroup`.

```
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
RadioButton radio3 = new RadioButton("Option 3");
ToggleGroup radioGroup = new ToggleGroup();
```

10. Assume a JavaFX application has a `RadioButton` control named `radio1`, and a `Label` control named `outputLabel`. Write a lambda expression that registers an event handler with the `RadioButton` control. The event handler should display the string “selected” in `outputLabel`.
11. Write code that creates a `ListView` control named `snackListView`. Add the following strings to the `Listview`: “cheese”, “olives”, “crackers”.
12. Write code that creates a `TextArea` displaying 10 rows and 15 columns. The text area should perform line wrapping.
13. Write code that creates a `Slider` control. The control should be horizontally oriented and its range should be 0 through 1000. Labels and tick marks should be displayed. Major tick marks should appear at every 100th number, and minor tick marks should appear at every 25th number. The initial value of the `Slider` should be set at 500.
14. Write the code that creates a menu bar with one menu named File. The

File menu should have the F key assigned as a mnemonic. The File menu should have three menu items: Open, Print, and Exit. Assign mnemonic keys of your choice to each of these items.

## Short Answer

1. In CSS, what is the difference between a type selector and an ID selector?
2. If you want to apply styles to all of the nodes in a scene, what selector would you use in a style definition?
3. You want the user to be able to select only one item from a group of items. Would you use RadioButton controls or CheckBox controls?
4. You want the user to be able to select any number of items from a group of items. Would you use RadioButton controls or CheckBox controls?
5. What is the purpose of a ToggleGroup?
6. Do you normally add RadioButton controls, CheckBox controls, or both to a ToggleGroup?
7. What is the difference between an editable ComboBox and an uneditable ComboBox?
8. What selection mode would you select if you want the user to only select a single item in a ListView?
9. You want to provide 20 items in a list for the user to select from. Which control would take up less space on the screen, a ListView or a ComboBox?

# Programming Challenges

## 1. Dorm and Meal Plan Calculator

A university has the following dormitories:



**VideoNote** The Dorm and Meal Plan Calculator Problem

- Allen Hall: \$1,800 per semester
- Pike Hall: \$2,200 per semester
- Farthing Hall: \$2,800 per semester
- University Suites: \$3,000 per semester

The university also offers the following meal plans:

- 7 meals per week: \$600 per semester
- 14 meals per week: \$1,100 per semester
- Unlimited meals: \$1,800 per semester

Create an application with two ComboBox controls. One should hold the names of the dormitories, and the other should hold the meal plans. The user should select a dormitory and a meal plan, and the application should show the total charges for the semester.

## 2. Skateboard Designer

The Skate Shop sells the skateboard products listed in [Table 12-9](#).

# Table 12-9 Skateboard products

| Decks                    | Truck Assemblies    | Wheels                   |
|--------------------------|---------------------|--------------------------|
| The Master Thrasher \$60 | 7.75-inch axle \$35 | 51 mm \$20               |
| The Dictator \$45        | 8-inch axle \$40    | 55 mm \$22               |
| The Street King \$50     | 8.5-inch axle \$45  | 58 mm \$24<br>61 mm \$28 |

In addition, the Skate Shop sells the following miscellaneous products:

- Grip tape: \$10
- Bearings: \$30
- Riser pads: \$2
- Nuts & bolts kit: \$3

Create an application that allows the user to select one deck, one truck assembly, and one wheel set from either ListView controls or ComboBox controls. The application should also have a ListView that allows the user to select multiple miscellaneous products. The application should display the subtotal, the amount of sales tax (at 7 percent), and the total of the order.

### 3. Conference Registration System

Create an application that calculates the registration fees for a conference. The general conference registration fee is \$895 per person, and student registration is \$495 per person. There is also an optional opening night dinner with a keynote speech for \$30 per person. In addition, the optional preconference workshops listed in [Table 12-10](#) are available.

# **Table 12-10 Optional preconference workshops**

| <b>Workshop</b>            | <b>Fee</b> |
|----------------------------|------------|
| Introduction to E-commerce | \$295      |
| The Future of the Web      | \$295      |
| Advanced Java Programming  | \$395      |
| Network Security           | \$395      |

The application should allow the user to select the registration type, the optional opening night dinner and keynote speech, and as many preconference workshops as desired. The total cost should be displayed.

## 4. Smartphone Packages

Cell Solutions, a cell phone provider, sells the following data plans:

- 8 gigabytes per month: \$45.00 per month
- 16 gigabytes per month: \$65.00 per month
- 20 gigabytes per month: \$99.00 per month

The provider sells the following phones. (A 6 percent sales tax applies to the sale of a phone.)

- Model 100: \$299.95
- Model 110: \$399.95
- Model 200: \$499.95

Customers may also select the following options:

- Phone Replacement Insurance: \$5.00 per month

- WiFi Hotspot Capability: \$10.00 per month

Write an application that displays a menu system. The menu system should allow the user to select one data plan, one phone, and any of the options desired. As the user selects items from the menu, the application should show the prices of the items selected.

## 5. Shopping Cart System

Create an application that works like a shopping cart system for an online book store. In the book’s source code (available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)) you will find a file named *BookPrices.txt*. This file contains the names and prices of various books, formatted in the following fashion:

- *I Did It Your Way*, 11.95
- *The History of Scotland*, 14.50
- *Learn Calculus in One Day*, 29.95
- *Feel the Stress*, 18.50

Each line in the file contains the name of a book, followed by a comma, followed by the book’s retail price. When your application begins execution, it should read the contents of the file and store the book titles in a `ListView`. The user should be able to select a title from the list and add it to a “shopping cart,” which is simply another `ListView` control. The application should have buttons or menu items that allow the user to remove items from the shopping cart, clear the shopping cart of all selections, and check out. When the user checks out, the application should calculate and display the subtotal of all the books in the shopping cart, the sales tax (which is 7 percent of the subtotal), and the total.

# **Chapter 13 JavaFX: Graphics, Effects, and Media**

## **Topics**

1. [13.1 Drawing Shapes](#)
2. [13.2 Animation](#)
3. [13.3 Effects](#)
4. [13.4 Playing Sound Files](#)
5. [13.5 Playing Videos](#)
6. [13.6 Handling Key Events](#)
7. [13.7 Handling Mouse Events](#)
8. [13.8 Common Errors to Avoid](#)

# 13.1 Drawing Shapes

## Concept:

JavaFX provides several classes that can be used to draw simple shapes, such as lines, circles, rectangles, and more.

JavaFX provides several classes for drawing simple 2D shapes. In this section, we will discuss classes for drawing lines, circles, rectangles, ellipses, arcs, polygons, polylines, and text. Before we examine how to draw these shapes, however, we must discuss the screen coordinate system. You use the *screen coordinate system* to specify the location of your graphics.



**VideoNote** Drawing Shapes with JavaFX

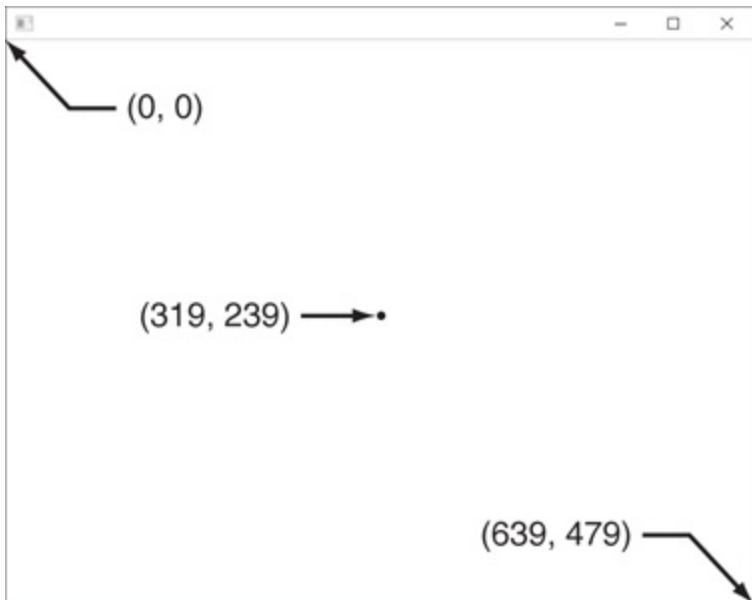
## The Screen Coordinate System

The images displayed on a computer screen are made up of tiny dots called *pixels*. The screen coordinate system is used to identify the position of each pixel in an application's window. Each pixel has an *X* coordinate and a *Y* coordinate. The *X* coordinate identifies the pixel's horizontal position, and the *Y* coordinate identifies its vertical position. The coordinates are usually written in the form  $(X, Y)$ . For example, the coordinates of the pixel in the upper-left corner of the screen are  $(0, 0)$ . This means that its *X* coordinate is 0, and its *Y* coordinate is 0.

The *X* coordinates increase from left to right, and the *Y* coordinates increase from top to bottom. In a window that is 640 pixels wide by 480 pixels high, the coordinates of the pixel at the bottom right corner of the window are  $(639,$

479). In the same window, the coordinates of the pixel in the center of the window are (319, 239). [Figure 13-1](#) shows the coordinates of various pixels in the window.

## Figure 13-1 Various pixel locations in a 640 by 480 window



### Note:

The screen coordinate system differs from the Cartesian coordinate system that you learned about in mathematics. In the Cartesian coordinate system, the Y coordinates decrease as you move downward. In the screen coordinate system, the Y coordinates increase as you move downward, toward the bottom of the screen.

# The Shape Class and Its Subclasses

The JavaFX library has a `Shape` class that provides the basic functionality for drawing shapes. The `Shape` class has several subclasses, each of which draws a specific shape. For example, there is a `Line` class for drawing lines, a `Circle` class for drawing circles, and so forth. [Table 13-1](#) lists some of subclasses of the `Shape` class.

## Table 13-1 Some of the Shape subclasses

| Class                  | Description                                                                                         |
|------------------------|-----------------------------------------------------------------------------------------------------|
| <code>Line</code>      | Draws a line from one point to another.                                                             |
| <code>Circle</code>    | Draws a circle with its center point located at a specific coordinate, and with a specified radius. |
| <code>Rectangle</code> | Draws a rectangle with a specified width and height.                                                |
| <code>Ellipse</code>   | Draws an ellipse with a specified center point, X radius, and Y radius.                             |
| <code>Arc</code>       | Draws an arc, which is a partial ellipse.                                                           |
| <code>Polygon</code>   | Draws a polygon with vertices at specified locations.                                               |
| <code>Polyline</code>  | Draws a polyline with vertices at specified locations.                                              |
| <code>Text</code>      | Draws a string at a specified location.                                                             |

The `Shape` class and its subclasses are in the `javafx.scene.shape` package. You draw shapes with these classes by following this general procedure:

1. Create an instance of the desired shape class (`Line`, `Circle`, and so forth), passing the necessary constructor arguments.
2. Repeat step 1 for each shape that you want to draw.
3. Add all of the shape objects that you created to a container.

#### 4. Add the container to the scene graph.

Notice in step 3 you add the shape objects to a container. You can use layout containers for this purpose, but we will use `Pane` objects because they are simple containers that do not perform layout. The `Pane` class is in the `javafx.scene.layout` package.

## The Line Class

The `Line` class has the two constructors that are summarized in [Table 13-2](#).

**Table 13-2 The Line class constructors**

| Constructor                                   | Description                                                                                                                                                                                                                                                           |
|-----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Line()</code>                           | Creates an empty line. Call the <code>Line</code> class's <code>setStartX</code> and <code>setStartY</code> methods to establish the line's starting point, and the <code>setEndX</code> and <code>setEndY</code> methods to establish the line's ending point        |
| <code>Line(startX, startY, endX, endY)</code> | All of the arguments are doubles. The <code>startX</code> and <code>startY</code> arguments are the X and Y coordinates for the line's starting point. The <code>endX</code> and <code>endY</code> arguments are the X and Y coordinates for the line's ending point. |

The second constructor is the one you will use most often. The following statement shows an example. It creates a line starting at the point (80, 120) and ending at the point (400, 520):

```
Line myLine = new Line(80, 120, 400, 520);
```

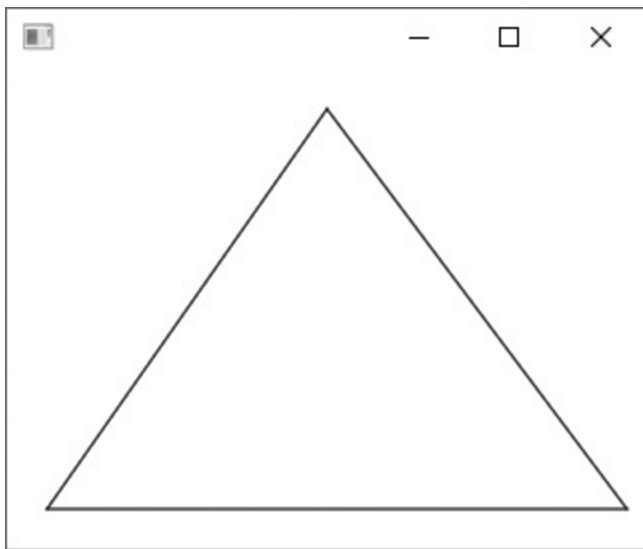
If you use the no-arg constructor, you have to use the class's `setStartX`, `setStartY`, `setEndX`, and `setEndY` methods to establish the line's starting and

ending points. Here is an example that creates a line starting at the point (0, 0) and ending at the point (200, 200):

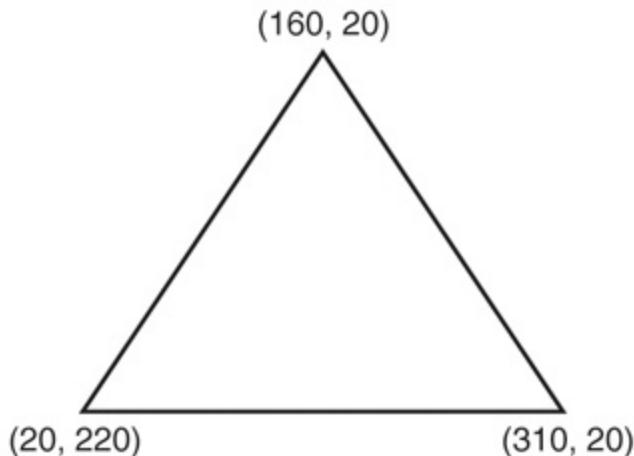
```
Line myLine = new Line();
myLine.setStartX(0);
myLine.setStartY(0);
myLine.setEndX(200);
myLine.setEndY(200);
```

[Code Listing 13-1](#) shows a program that uses the `Line` class to draw a triangle. The programs output is shown in [Figure 13-2](#). [Figure 13-3](#) shows the coordinates of the triangle's corners.

## Figure 13-2 Output of Triangle.java



## Figure 13-3 Coordinates of the triangle's corners



## Code Listing 13-1 (Triangle.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Line;
6
7 public class Triangle extends Application
8 {
9 public static void main(String[] args)
10 {
11 launch(args);
12 }
13
14 @Override
15 public void start(Stage primaryStage)
16 {
17 // Constants for the scene size
18 final double SCENE_WIDTH = 320.0;
19 final double SCENE_HEIGHT = 240.0;
20
21 // Constants for the triangle corners.
22 final double TOP_X = 160.0;
23 final double TOP_Y = 20.0;
24 final double BOTTOM_RIGHT_X = 310.0;
25 final double BOTTOM_RIGHT_Y = 220.0;
26 final double BOTTOM_LEFT_X = 20.0;
27 final double BOTTOM_LEFT_Y = 220.0;
28
29 // Draw the first line, from the top of the triangle
```

```

30 // to the bottom-right corner.
31 Line line1 = new Line(TOP_X, TOP_Y, BOTTOM_RIGHT_X, BOTTOM_
32
33 // Draw the second line, from the top of the triangle
34 // to the bottom-left corner.
35 Line line2 = new Line(TOP_X, TOP_Y, BOTTOM_LEFT_X, BOTTOM_L
36
37 // Draw the third line, from the bottom-left corner
38 // of the triangle to the bottom-right corner.
39 Line line3 = new Line(BOTTOM_LEFT_X, BOTTOM_LEFT_Y,
40 BOTTOM_RIGHT_X, BOTTOM_RIGHT_Y);
41
42 // Add the lines to a Pane.
43 Pane pane = new Pane(line1, line2, line3);
44
45 // Create a Scene and display it.
46 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
47 primaryStage.setScene(scene);
48 primaryStage.show();
49 }
50 }
```

Let's take a closer look at the program:

- Lines 18 and 19 define constants for the scene's width and height.
- Lines 22 and 23 define the constants `TOP_X` and `TOP_Y`, that are used as the  $(X,Y)$  coordinates for the top corner of the triangle.
- Lines 24 and 25 define the constants `BOTTOM_RIGHT_X` and `BOTTOM_RIGHT_Y`, that are used as the  $(X,Y)$  coordinates for the bottom-right corner of the triangle.
- Lines 26 and 27 define the constants `BOTTOM_LEFT_X` and `BOTTOM_LEFT_Y`, that are used as the  $(X,Y)$  coordinates for the bottom-left corner of the triangle.
- The statement in line 31 creates a line that starts at the top corner of the triangle (160, 20) and ends at the bottom-right corner of the triangle (310, 220).
- The statement in line 35 creates a line that starts at the top corner of the triangle (160, 20) and ends at the bottom-left corner of the triangle (20,

220).

- The statement in lines 39 and 40 creates a line that starts at the bottom-left corner of the triangle (20, 220) and ends at the bottom-right corner of the triangle (310, 220).
- The statement in line 43 creates a Pane object and adds the three Line objects to it.
- The statement in line 46 creates a Scene object containing the Pane. The scene is 320 pixels wide by 240 pixels high.
- The statement in line 47 sets the scene to the stage.
- The statement in line 48 displays the application's window.

## Changing the Stroke Color

The default color of lines and other shapes is black. To change a shape's color to a different value, call the `setStroke` method, which is inherited from the `Shape` class. The method's general format is

`setStroke(color)`

where the `color` argument is an object of the `Paint` class. Usually, you will pass one of the `Color` class's constants as an argument. The `Color` class, which is in the `javafx.scene.paint` package, defines several static constant fields that represent standard colors. Some examples are `Color.WHITE`, `Color.RED`, `Color.BLUE`, `Color.BLACK`, and so forth. (There are constants for other, more interesting, colors as well, such as `Color.ALICEBLUE` and `Color.BLANCHEDALMOND`. For a complete list of the fields, see the Java API documentation for the `Color` class.)

The following code snippet shows an example of creating a `Line` object and setting its stroke color to red:

```
Line myLine = new Line(80, 120, 400, 520);
```

```
myLine.setStroke(Color.RED);
```

## Tip:

Remember, to use the `Color` class, you must import it with the following statement:

```
import javafx.scene.paint.Color;
```

# The `Circle` Class

The `Circle` class has five constructors, summarized in [Table 13-3](#).

**Table 13-3 The `circle` class constructors**

| Constructor                                   | Description                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Circle()</code>                         | Creates an empty circle. Call the <code>circle</code> class's <code>setCenterX</code> and <code>setCenterY</code> methods to establish the circle's center point, the <code>setRadius</code> method to establish the circle's radius, and the <code>setFill</code> method to fill the circle with a color. |
| <code>Circle(radius)</code>                   | The <code>radius</code> argument is a double. Creates a circle of the specified radius. Call the <code>circle</code> class's <code>setCenterX</code> and <code>setCenterY</code> methods to establish the circle's center point, and the <code>setFill</code> method to fill the circle with a color.      |
| <code>Circle(centerX, centerY, radius)</code> | The arguments are doubles. Creates a circle at the specified center point, with the specified radius. By default, the circle is filled with the color black. Call the <code>circle</code> class's <code>setFill</code> method to fill the circle with a different color.                                   |

The `centerX`, `centerY`, and `radius` arguments are doubles, and the `fill` argument is a `Paint` class object (typically a `Color` class constant).  
`Circle(centerX, centerY, radius, fill)` Creates a circle at the specified center point, with the specified radius, and filled with the specified color.

The `radius` argument is a double, and the `fill` argument is a `Paint` class object (typically a `Color` class constant). Creates a circle with the specified radius, and filled with the specified color. Call the `Circle` class's `setCenterX` and `setCenterY` methods to establish the circle's center point, and the `setRadius` method to establish the circle's radius.

The following statement creates a circle with its center point at (200, 200) and with a radius of 150:

```
Circle myCircle = new Circle(200, 200, 150);
```

You can use the `Circle` class's `setCenterX`, `setCenterY`, `setRadius`, and `setFill` methods to establish the circle's center point, radius, and fill color. Here is an example that creates a circle with its center point at (75, 100), a radius of 50, and filled with the color red:

```
Circle myCircle = new Circle();
myCircle.setCenterX(75);
myCircle.setCenterY(100);
myCircle.setRadius(50);
myCircle.setFill(Color.RED);
```

If you prefer the circle to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `setStroke` method to change the color of the circle's outline. Here is an example that draws a black outline of a circle with no fill color:

```
Circle myCircle = new Circle(75, 100, 50);
myCircle.setFill(null);
myCircle.setStroke(Color.BLACK);
```

[Code Listing 13-2](#) shows a program that uses the `Circle` class to draw a bull's eye. The bull's eye is made of four concentric circles filled with the alternating colors black and red. The program's output is shown in [Figure 13-4](#).

## Figure 13-4 Output of BullsEye.java



## Code Listing 13-2 (BullsEye.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Circle;
6 import javafx.scene.paint.Color;
7
8 public class BullsEye extends Application
9 {
10 public static void main(String[] args)
11 {
12 launch(args);
```

```

13 }
14
15 @Override
16 public void start(Stage primaryStage)
17 {
18 // Constants for the scene size
19 final double SCENE_WIDTH = 640.0;
20 final double SCENE_HEIGHT = 480.0;
21
22 // Constants for the bull's eye center point and radii.
23 final double CENTER_X = 320.0;
24 final double CENTER_Y = 240.0;
25 final double RING1_RADIUS = 240.0;
26 final double RING2_RADIUS = 150.0;
27 final double RING3_RADIUS = 75.0;
28 final double RING4_RADIUS = 25.0;
29
30 // Create the circles.
31 Circle ring1 = new Circle(CENTER_X, CENTER_Y,
32 RING1_RADIUS, Color.BLACK);
33
34 Circle ring2 = new Circle(CENTER_X, CENTER_Y,
35 RING2_RADIUS, Color.RED);
36
37 Circle ring3 = new Circle(CENTER_X, CENTER_Y,
38 RING3_RADIUS, Color.BLACK);
39
40 Circle ring4 = new Circle(CENTER_X, CENTER_Y,
41 RING4_RADIUS, Color.RED);
42
43 // Add the circles to a Pane.
44 Pane pane = new Pane(ring1, ring2, ring3);
45
46 // Create a Scene and display it.
47 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
48 primaryStage.setScene(scene);
49 primaryStage.show();
50 }
51 }
```

Let's take a closer look at the program:

- Lines 19 and 20 define constants for the scene's width and height.
- Lines 23 and 24 define the constants `CENTER_X` and `CENTER_Y`, that are used as the  $(X,Y)$  coordinates for the center of the bull's eye.

- Lines 25 through 28 define the constants RING1\_RADIUS, RING2\_RADIUS, RING3\_RADIUS, and RING4\_RADIUS. These values are the radii of the four circles.
- Lines 31 and 32 create the first circle, filled with the color black.
- Lines 34 and 35 create the second circle, on top of the first circle, filled with the color red.
- Lines 37 and 38 create the third circle, on top of the previous circles, filled with the color black.
- Lines 40 and 41 create the fourth circle, on top of the previous circles, filled with the color red.
- The statement in line 44 creates a Pane object and adds the four Circle objects to it.
- The statement in line 47 creates a Scene object containing the Pane. The scene is 640 pixels wide by 480 pixels high.
- The statement in line 48 sets the scene to the stage.
- The statement in line 49 displays the application's window.

## The Rectangle Class

The Rectangle class has four constructors, summarized in [Table 13-4](#).

**Table 13-4 The Rectangle class constructors**

| Constructor | Description                          |
|-------------|--------------------------------------|
|             | Creates an empty rectangle. Call the |

|                                             |                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Rectangle()</code>                    | Rectangle class's <code>setX</code> and <code>setY</code> methods to establish the position of the rectangle's upper-left corner, the <code>setWidth</code> method to establish the rectangle's width, the <code>setHeight</code> method to establish the rectangle's height, and the <code>setFill</code> method to fill the rectangle with a color. (By default, the rectangle is filled with black.) |
| <code>Rectangle(width, height)</code>       | The arguments are doubles. Creates a rectangle of the specified width and height. Call the Rectangle class's <code>setX</code> and <code>setY</code> methods to establish the position of the rectangle's upper-left corner, and the <code>setFill</code> method to fill the rectangle with a color. (By default, the rectangle is filled with black.)                                                  |
| <code>Rectangle(x, y, width, height)</code> | The arguments are doubles. Creates a rectangle with its upper-left corner at the specified <i>(X,Y)</i> coordinates, and of the specified width and height. Call the Rectangle class's <code>setFill</code> method to fill the rectangle with a color. (By default, the rectangle is filled with black.)                                                                                                |
| <code>Rectangle(width, height, fill)</code> | The <i>width</i> and <i>height</i> arguments are doubles, and the <i>fill</i> argument is a Paint class object (typically a Color class constant). Creates a rectangle of the specified width and height, and filled with the specified color. Call the Rectangle class's <code>setX</code> and <code>setY</code> methods to establish the position of the rectangle's upper-left corner.               |

The following statement creates a rectangle with its upper-left corner at (200, 100), with a width of 75 and a height of 150:

```
Rectangle myRectangle = new Rectangle(200, 100, 75, 150);
```

You can use the Rectangle class's `setX` and `setY` methods to establish the

position of the rectangle's upper-left corner, the `setWidth` method to establish the rectangle's width, the `setHeight` method to establish the rectangle's height, and the `setFill` method to fill the rectangle with a specific color. Here is an example that creates a rectangle with its upper-left corner at (10, 20), a width of 50, a height of 100, and filled with the color dark green:

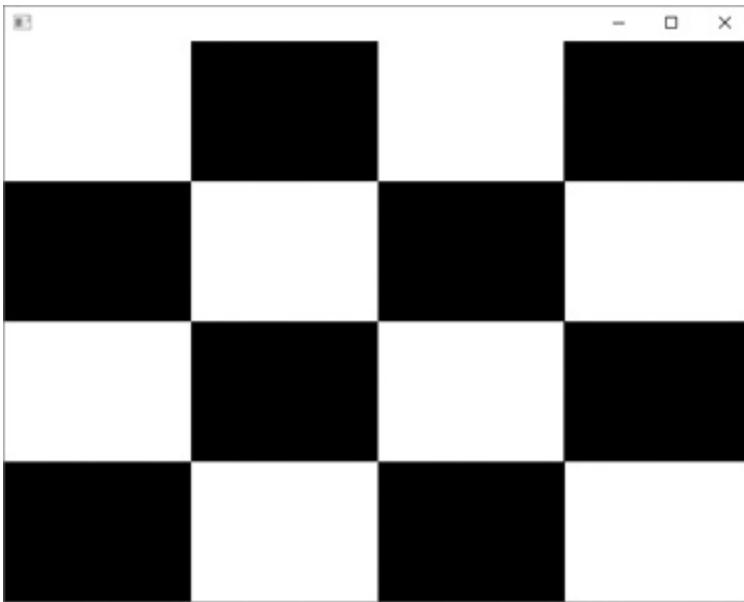
```
Rectangle myRectangle = new Rectangle();
myRectangle.setX(10);
myRectangle.setY(20);
myRectangle.setWidth(50);
myRectangle.setHeight(100);
myRectangle.setFill(Color.DARKGREEN);
```

If you prefer the rectangle to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `setStroke` method to change the color of the rectangle's outline. Here is an example that draws a black outline of a rectangle with no fill color:

```
Rectangle myRectangle = new Rectangle(200, 100, 75, 150);
myRectangle.setFill(null);
myRectangle.setStroke(Color.BLACK);
```

[Code Listing 13-3](#) demonstrates how the `Rectangle` class can be used to create a checkerboard pattern. The program creates eight rectangles of the same size, filled with the default color black. The rectangles are displayed in a checkerboard pattern on a white background. The program's output is shown in [Figure 13-5](#).

## Figure 13-5 Output of CheckerBoard.java



## Code Listing 13-3 (**CheckerBoard.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Rectangle;
6
7 public class CheckerBoard extends Application
8 {
9 public static void main(String[] args)
10 {
11 launch(args);
12 }
13
14 @Override
15 public void start(Stage primaryStage)
16 {
17 // Constants for the scene size
18 final double SCENE_WIDTH = 640.0;
19 final double SCENE_HEIGHT = 480.0;
20
21 // Constants for all box's width and height
22 final double WIDTH = 160.0;
23 final double HEIGHT = 120.0;
```

```
24
25 // Constants for each box's (X,Y) coordinates
26 final double BOX1_X = 160.0;
27 final double BOX1_Y = 0.0;
28
29 final double BOX2_X = 479.0;
30 final double BOX2_Y = 0.0;
31
32 final double BOX3_X = 0.0;
33 final double BOX3_Y = 120.0;
34
35 final double BOX4_X = 320.0;
36 final double BOX4_Y = 120.0;
37
38 final double BOX5_X = 160.0;
39 final double BOX5_Y = 240.0;
40
41 final double BOX6_X = 479.0;
42 final double BOX6_Y = 240.0;
43
44 final double BOX7_X = 0.0;
45 final double BOX7_Y = 360.0;
46
47 final double BOX8_X = 320.0;
48 final double BOX8_Y = 360.0;
49
50 // Create the boxes.
51 Rectangle box1 = new Rectangle(BOX1_X, BOX1_Y, WIDTH, HEIGHT);
52 Rectangle box2 = new Rectangle(BOX2_X, BOX2_Y, WIDTH, HEIGHT);
53 Rectangle box3 = new Rectangle(BOX3_X, BOX3_Y, WIDTH, HEIGHT);
54 Rectangle box4 = new Rectangle(BOX4_X, BOX4_Y, WIDTH, HEIGHT);
55 Rectangle box5 = new Rectangle(BOX5_X, BOX5_Y, WIDTH, HEIGHT);
56 Rectangle box6 = new Rectangle(BOX6_X, BOX6_Y, WIDTH, HEIGHT);
57 Rectangle box7 = new Rectangle(BOX7_X, BOX7_Y, WIDTH, HEIGHT);
58 Rectangle box8 = new Rectangle(BOX8_X, BOX8_Y, WIDTH, HEIGHT);
59
60 // Add the boxes to a Pane.
61 Pane pane = new Pane(box1, box2, box3, box4,
62 box5, box6, box7, box8);
63
64 // Create a Scene and display it.
65 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
66 primaryStage.setScene(scene);
67 primaryStage.show();
68 }
69 }
```

Let's take a closer look at the program:

- Lines 18 and 19 define constants for the scene's width and height.
- Lines 22 and 23 define the constants `WIDTH` and `HEIGHT`, which are used as the width and height for each box that makes up the checkerboard.
- Lines 26 through 48 define constants for the  $(X, Y)$  coordinates for the upper-left corner of each rectangle.
- Lines 51 through 58 create the eight `Rectangle` objects. By default, they are filled with the color black.
- The statement in lines 61 and 62 creates a `Pane` object and adds the eight `Rectangle` objects to it.
- The statement in line 65 creates a `Scene` object containing the `Pane`. The scene is 640 pixels wide by 480 pixels high.
- The statement in line 66 sets the scene to the stage.
- The statement in line 67 displays the application's window.

## The Ellipse Class

The `Ellipse` class draws an ellipse, which is an oval shape. The `Ellipse` class has three constructors, summarized in [Table 13-5](#).

### Table 13-5 The `Ellipse` class constructors

| Constructor | Description                                                                                                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------|
|             | Creates an empty ellipse. Call the <code>Ellispe</code> class's <code>setCenterX</code> and <code>setCenterY</code> methods |

`Ellipse()`

to establish the ellipse's center point, the `setRadiusX` method to establish the ellipse's radius along the *X* axis, and the `setRadiusY` method to establish the ellipse's radius along the *Y* axis.

`Ellipse(radiusX,  
radiusY)`

The arguments are doubles. Creates an ellipse with the specified radii along the *X* and *Y* axes. Call the `Ellispe` class's `setCenterX` and `setCenterY` methods to establish the ellipse's center point.

`Ellipse(centerX,  
centerY,  
radiusX,  
radiusY)`

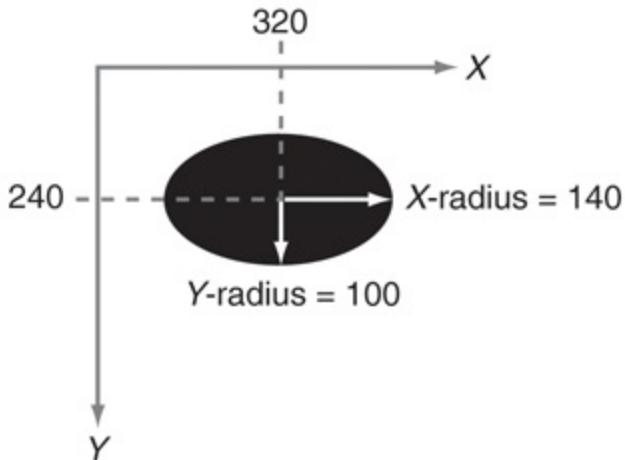
The arguments are doubles. Creates an ellipse at the specified center point, with the specified *X* and *Y* radii.

Here is an example:

```
Ellipse myEllipse = new Ellipse(320, 240, 140, 100);
```

As shown in [Figure 13-6](#), the ellipse created by this statement will have its center located at (320, 240), an *X*-radius of 140 pixels, and a *Y*-radius of 100. The use of two different radii, one for the *X* axis and one for the *Y* axis, gives the ellipse its shape.

## Figure 13-6 An ellipse's center point, x-radius, and y-radius



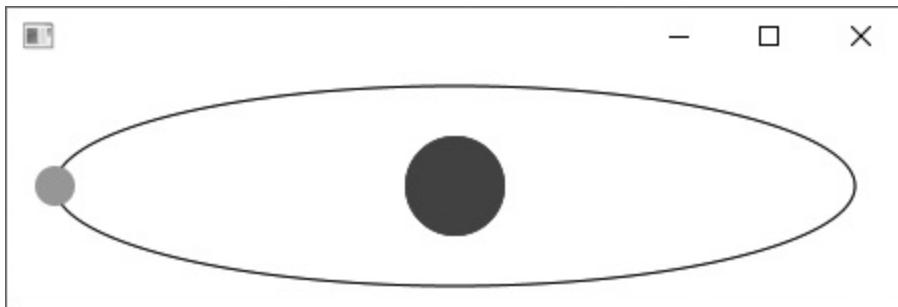
You can use the `Ellipse` class's `setCenterX`, `setCenterY`, `setRadiusX`, and `setRadiusY` methods to establish the ellipse's center point, *X*-radius, and *Y*-radius. By default, the ellipse will be filled with the color black. You can call the `Ellipse` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the ellipse to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Ellipse` class's `setStroke` method (also inherited from the `Shape` class) to change the color of the ellipse's outline.

The following code shows an example. It creates an ellipse with its center point at (125, 100), an *X*-radius of 130, a *Y*-radius of 90, no fill color, and a stroke color of black:

```
Ellipse myEllipse = new Ellipse();
myEllipse.setCenterX(125);
myEllipse.setCenterY(100);
myEllipse.setRadiusX(130);
myEllipse.setRadiusY(90);
myEllipse.setFill(null);
myellipse.setStroke(Color.BLACK);
```

The program shown in [Code Listing 13-4](#) draws a blue circle representing a planet, and a gray circle representing a moon. An ellipse is also drawn representing the moon's orbital path around the planet. The program's output is shown in [Figure 13-7](#).

# Figure 13-7 Output of Orbit.java



## Code Listing 13-4 (Orbit.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.paint.Color;
6 import javafx.scene.shape.Circle;
7 import javafx.scene.shape.Ellipse;
8
9 public class Orbit extends Application
10 {
11 public static void main(String[] args)
12 {
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Constants for the scene size
20 final double SCENE_WIDTH = 450.0;
21 final double SCENE_HEIGHT = 120.0;
22
23 // Constants for the planet, orbit, and moon
24 final double PLANET_X = 224.0;
25 final double PLANET_Y = 59.0;
26 final double PLANET_RAD = 25.0;
27 }
}
```

```

28 final double ORBIT_RAD_X = 200.0;
29 final double ORBIT_RAD_Y = 50.0;
30
31 final double MOON_X = 24.0;
32 final double MOON_Y = 59.0;
33 final double MOON_RAD = 10.0;
34
35 // Draw the planet.
36 Circle planet = new Circle(PLANET_X, PLANET_Y,
37 PLANET_RAD, Color.BLUE);
38
39 // Draw the elliptical orbit path.
40 Ellipse orbitPath = new Ellipse(PLANET_X, PLANET_Y,
41 ORBIT_RAD_X, ORBIT_RAD_Y);
42 orbitPath.setFill(null);
43 orbitPath.setStroke(Color.BLACK);
44
45 // Draw the moon.
46 Circle moon = new Circle(MOON_X, MOON_Y,
47 MOON_RAD, Color.DARKGRAY);
48
49 // Add the boxes to a Pane.
50 Pane pane = new Pane(planet, orbitPath, moon);
51
52 // Create a Scene and display it.
53 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
54 primaryStage.setScene(scene);
55 primaryStage.show();
56 }
57 }
```

Let's take a closer look at the program:

- Lines 20 and 21 define constants for the scene's width and height.
- Lines 24 through 26 define constants for the planet's center point and radius.
- Lines 28 and 29 define constants for the orbit's *X*-radius and *Y*-radius.
- Lines 31 through 33 define constants for the moon's center point and radius.
- Lines 36 and 37 create the planet as a circle filled with the color blue.

- Lines 40 and 41 create the orbital path as an ellipse. The center point of the ellipse is the same as the center point of the planet.
- Line 42 sets the ellipse's fill color to `null`. As a result, the ellipse will not be filled with a color.
- Line 43 sets the ellipse's stroke color to black.
- Lines 46 and 47 create the moon as a circle filled with the color dark gray.
- Line 50 creates a `Pane`, and adds the `planet`, `orbitPath`, and `moon` objects to it.
- Line 53 creates a `Scene` object containing the `Pane`. The scene is 450 pixels wide by 120 pixels high.
- Line 54 sets the scene to the stage.
- Line 55 displays the application's window.



## Note:

If an ellipse's X-radius and Y-radius have the same value, then the ellipse will be a circle.

## The Arc Class

The `Arc` class draws an arc, which is considered to be part of an oval. The `Arc` class has two constructors, summarized in [Table 13-6](#).

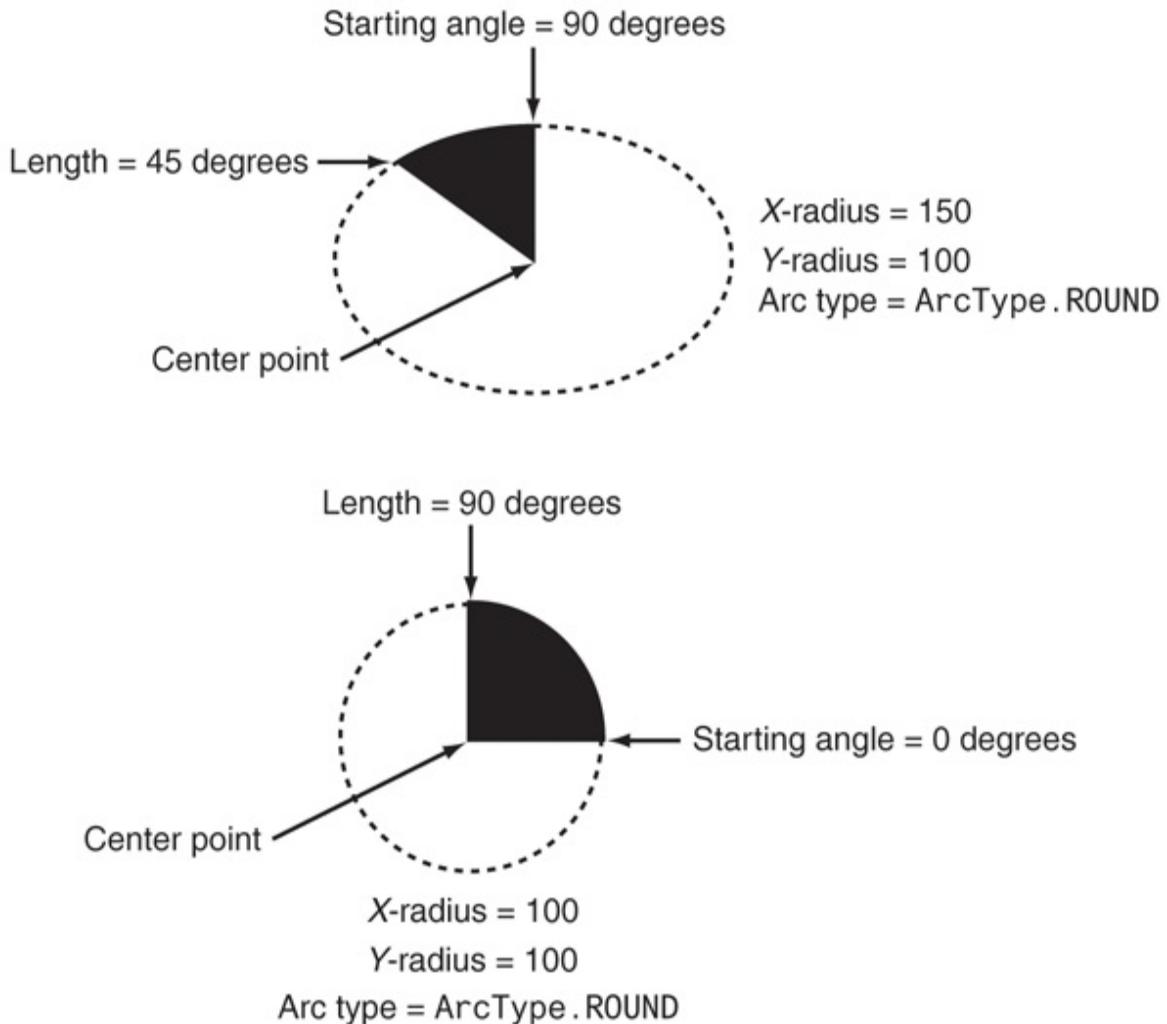
## Table 13-6 The Arc class

# constructors

| Constructor                                                                                                                 | Description                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Arc()                                                                                                                       | Creates an empty arc. Call the Arc class's setCenterX and setCenterY methods to establish the arc's center point, the setRadiusX method to establish the arc's radius along the X axis, the setRadiusY method to establish the arc's radius along the Y axis, the setStartAngle method to establish the arc's starting angle (in degrees), and the setLength method to establish the arc's angular extent (in degrees). |
| Arc( <i>centerX</i> ,<br><i>centerY</i> ,<br><i>radiusX</i> ,<br><i>radiusY</i> ,<br><i>startAngle</i> ,<br><i>length</i> ) | The arguments are doubles. Creates an arc at the specified center point, with the specified X and Y radii. The arc begins at the angle specified by <i>startAngle</i> , and extends counterclockwise. The <i>length</i> argument specifies the number of degrees that the arc extends from its starting angle.                                                                                                          |

Because an arc is part of an ellipse, you need to establish a center point, an X-radius, and a Y-radius. In addition, an arc has a starting angle and an angular length, both of which are specified in degrees. Note 0 degrees is at the 3 o'clock position, and arcs are drawn counterclockwise from their starting angle. [Figure 13-8](#) summarizes the properties of an arc with two examples.

## Figure 13-8 Arc properties



[Figure 13-8 Full Alternative Text](#)

By default, the arc will be filled with the color black. You can call the `Arc` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the arc to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Arc` class's `setStroke` method (also inherited from the `Shape` class) to change the color of the ellipse's outline.

There are three types of arc that you can draw with the `Arc` class, summarized in [Table 13-7](#). To select one of these arc types, you must import the `ArcType` class from the `javafx.scene.shape` package. You then pass one of the constants listed in [Table 13-7](#) to the `Arc` class's `setType` method. (Note the default type is `ArcType.CHORD`.) [Figure 13-9](#) shows examples of each type of arc.

## Table 13-7 Arc types

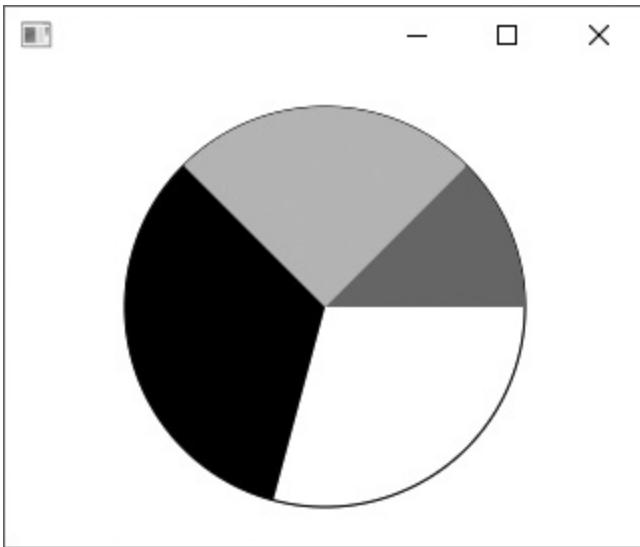
| Type          | Description                                                                                                                      |
|---------------|----------------------------------------------------------------------------------------------------------------------------------|
| ArcType.CHORD | This is the default arc type. A straight line will be drawn from one endpoint of the arc to the other endpoint.                  |
| ArcType.ROUND | Straight lines will be drawn from each endpoint to the arc's center point. As a result, the arc will be shaped like a pie slice. |
| ArcType.OPEN  | No lines will connect the endpoints. Only the arc will be drawn.                                                                 |

## Figure 13-9 Types of arcs



[Code Listing 13-5](#) shows an example program that uses arcs to draw a pie chart. The program's output is shown in [Figure 13-10](#).

## Figure 13-10 Output of PieChart.java



## Code Listing 13-5 (PieChart.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.paint.Color;
6 import javafx.scene.shape.Circle;
7 import javafx.scene.shape.Arc;
8 import javafx.scene.shape.ArcType;
9
10 public class Arcs extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Constants for the scene size
21 final double SCENE_WIDTH = 320.0;
22 final double SCENE_HEIGHT = 240.0;
23
24 // Common attributes for the arcs
25 final double X = 160.0; // Center point X coordinate
26 final double Y = 120.0; // Center point Y coordinate
27 final double X_RAD = 100.00; // X-radius
```

```

28 final double Y_RAD = 100.0; // Y-radius
29
30 // Constants for the arc angles
31 final double PIE1_START = 0.0;
32 final double PIE1_LENGTH = 45.0;
33 final double PIE2_START = 45.0;
34 final double PIE2_LENGTH = 90.0;
35 final double PIE3_START = 135.0;
36 final double PIE3_LENGTH = 120.0;
37
38 // Draw a circle outline.
39 Circle outline = new Circle(X, Y, X_RAD, Color.WHITE);
40 outline.setStroke(Color.BLACK);
41
42 // Draw the first pie-slice.
43 Arc pieSlice1 = new Arc(X,Y, X_RAD, Y_RAD,
44 PIE1_START, PIE1_LENGTH);
45 pieSlice1.setFill(Color.RED);
46 pieSlice1.setType(ArcType.ROUND);
47
48 // Draw the second pie-slice.
49 Arc pieSlice2 = new Arc(X,Y, X_RAD, Y_RAD,
50 PIE2_START, PIE2_LENGTH);
51 pieSlice2.setFill(Color.LIGHTGREEN);
52 pieSlice2.setType(ArcType.ROUND);
53
54 // Draw the third pie-slice.
55 Arc pieSlice3 = new Arc(X,Y, X_RAD, Y_RAD,
56 PIE3_START, PIE3_LENGTH);
57 pieSlice3.setFill(Color.BLACK);
58 pieSlice3.setType(ArcType.ROUND);
59
60 // Add the objects to a Pane.
61 Pane pane = new Pane(outline, pieSlice1, pieSlice2,
62 pieSlice3);
63
64 // Create a Scene and display it.
65 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
66 primaryStage.setScene(scene);
67 primaryStage.show();
68 }
69 }
```

Let's take a closer look at the program:

- Lines 21 and 22 define constants for the scene's width and height.

- Lines 25 through 28 define constants for the common attributes of the arcs (center point coordinates, X-radius, and Y-radius).
- Lines 31 through 36 define constants for each of the pie slice’s starting angle and angular length.
- Lines 39 and 40 create a white circle with a black outline. This will serve as the outline for the “pie.”
- Lines 43 through 46 create the first pie slice, sets its fill color to red, and its type to `ArcType.ROUND`.
- Lines 49 through 52 create the second pie slice, sets its fill color to light green, and its type to `ArcType.ROUND`.
- Lines 55 through 58 create the third pie slice, sets its fill color to black, and its type to `ArcType.ROUND`.
- Lines 61 and 62 create a `Pane`, and adds the `outline`, `pieSlice1`, `pieSlice2`, and `pieSlice3` objects to it.
- Line 65 creates a `Scene` object containing the `Pane`. The scene is 320 pixels wide by 240 pixels high.
- Line 66 sets the scene to the stage.
- Line 67 displays the application’s window.

## The Polygon Class

The `Polygon` class creates a closed polygon that, by default, is filled with the color black. A polygon is constructed of multiple line segments that are connected. The point where two line segments are connected is called a *vertex*. The `Polygon` class has two constructors, summarized in [Table 13-8](#).

## Table 13-8 The Arc class

# constructors

| Constructor                                                                                               | Description                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Polygon()                                                                                                 | Creates an empty polygon. Call the <code>Polygon</code> class's <code>getPoints().addAll()</code> method to establish the <i>XY</i> coordinates of the polygon's vertices. |
| Polygon( <i>x<sub>1</sub></i> , <i>y<sub>1</sub></i> , <i>x<sub>2</sub></i> , <i>y<sub>2</sub></i> , ...) | The variable number of arguments, which are doubles, are the <i>XY</i> coordinates of the polygon's vertices.                                                              |

When you call the parameterized constructor, you pass the (*X,Y*) coordinates of each vertex as arguments. The first two arguments are the (*X,Y*) coordinates of the first vertex, the next two arguments are the (*X,Y*) coordinates of the second vertex, and so forth. Here is an example that draws a diamond shape:

```
Polygon diamond = new Polygon(160.0, 20.0, // Top
 300.0, 120.0, // Right
 160.0, 220.0, // Bottom
 20.0, 120.0, // Left
 160.0, 20.0); // Top
```

If you call the no-arg constructor to create an empty polygon, you can then call the `Polygon` class's `getPoints().addAll()` method to establish the coordinates of the vertices. Here is an example that creates the diamond shape previously shown:

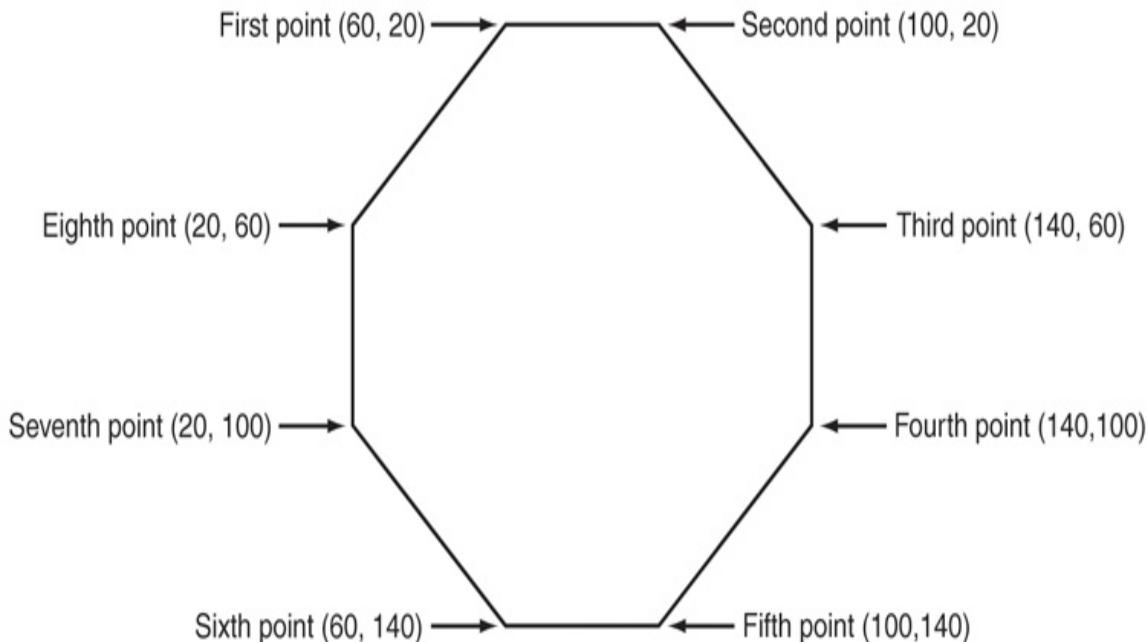
```
Polygon diamond = new Polygon();
diamond.getPoints().addAll(160.0, 20.0, // Top
 300.0, 120.0, // Right
 160.0, 220.0, // Bottom
 20.0, 120.0, // Left
 160.0, 20.0); // Top
```

By default, polygons are filled with the color black. You can call the `Polygon` class's `setFill` method (inherited from the `Shape` class) to fill it with another color. If you prefer the polygon to not be filled, pass `null` as an argument to the `setFill` method. Then, call the `Polygon` class's `setStroke` method (also

inherited from the Shape class) to change the color of the polygon outline.

[Code Listing 13-6](#) shows an example program that uses the Polygon class to draw a red octagon with the vertices shown in [Figure 13-11](#). The program's output is shown in [Figure 13-12](#).

## Figure 13-11 Vertices of the octagon displayed by Octagon.java



## Code Listing 13-6 (Octagon.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.paint.Color;
6 import javafx.scene.shape.Polygon;
7
```

```

8 public class Octagon extends Application
9 {
10 public static void main(String[] args)
11 {
12 launch(args);
13 }
14
15 @Override
16 public void start(Stage primaryStage)
17 {
18 // Constants for the scene size
19 final double SCENE_WIDTH = 160.0;
20 final double SCENE_HEIGHT = 160.0;
21
22 // Constants for the vertices.
23 final double X1 = 60.0, Y1 = 20.0;
24 final double X2 = 100.0, Y2 = 20.0;
25 final double X3 = 140.0, Y3 = 60.0;
26 final double X4 = 140.0, Y4 = 100.0;
27 final double X5 = 100.0, Y5 = 140.0;
28 final double X6 = 60.0, Y6 = 140.0;
29 final double X7 = 20.0, Y7 = 100.0;
30 final double X8 = 20.0, Y8 = 60.0;
31
32 // Create an octagon.
33 Polygon octagon = new Polygon(X1, Y1, X2, Y2,
34 X3, Y3, X4, Y4,
35 X5, Y5, X6, Y6,
36 X7, Y7, X8, Y8);
37
38 // Set the octagon's fill color to red.
39 octagon.setFill(Color.RED);
40
41 // Add the octagon to a Pane.
42 Pane pane = new Pane(octagon);
43
44 // Create a Scene and display it.
45 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
46 primaryStage.setScene(scene);
47 primaryStage.show();
48 }
49 }

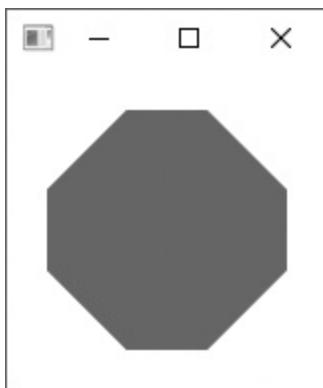
```

Let's take a closer look at the program:

- Lines 19 and 20 define constants for the scene's width and height.

- Lines 23 through 30 define constants for the *X* and *Y* coordinates of each vertex.
- Lines 33 through 36 create the octagon.
- Line 39 sets the octagon's fill color to `Color.RED`.
- Line 42 creates a `Pane`, and adds the octagon object to it.
- Line 45 creates a `Scene` object containing the `Pane`. The scene is 160 pixels wide by 160 pixels high.
- Line 46 sets the scene to the stage.
- Line 47 displays the application's window.

## Figure 13-12 Output of Octogon.java



## The Polyline Class

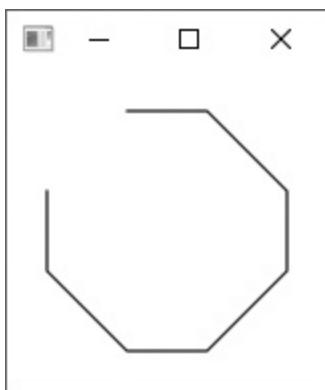
The `Polyline` class creates a line with multiple segments. A polyline is very similar to a polygon, except that a polyline is not automatically closed. The `Polyline` class has two constructors, summarized in [Table 13-9](#).

## Table 13-9 The PolyLine class constructors

| Constructor                   | Description                                                                                                                                  |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| Polyline()                    | Creates an empty polyline. Call the Polyline class's getPoints().addAll() method to establish the XY coordinates of the polyline's vertices. |
| Polyline(x1, y1, x2, y2, ...) | The variable number of arguments, which are doubles, are the XY coordinates of the polyline's vertices.                                      |

To demonstrate the similarities between the Polyline class and the Polygon class, look at [Code Listing 13-7](#). It creates a polyline using the same vertex coordinates as the Octagon.java program in [Code Listing 13-6](#). The program's output is shown in [Figure 13-13](#).

## Figure 13-13 Output of PolylineDemo.java



## Code Listing 13-7

# (PolylineDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Polyline;
6
7 public class PolylineDemo extends Application
8 {
9 public static void main(String[] args)
10 {
11 launch(args);
12 }
13
14 @Override
15 public void start(Stage primaryStage)
16 {
17 // Constants for the scene size
18 final double SCENE_WIDTH = 160.0;
19 final double SCENE_HEIGHT = 160.0;
20
21 // Constants for the vertices.
22 final double X1 = 60.0, Y1 = 20.0;
23 final double X2 = 100.0, Y2 = 20.0;
24 final double X3 = 140.0, Y3 = 60.0;
25 final double X4 = 140.0, Y4 = 100.0;
26 final double X5 = 100.0, Y5 = 140.0;
27 final double X6 = 60.0, Y6 = 140.0;
28 final double X7 = 20.0, Y7 = 100.0;
29 final double X8 = 20.0, Y8 = 60.0;
30
31 // Create a polyline.
32 Polyline polyline = new Polyline(X1, Y1, X2, Y2,
33 X3, Y3, X4, Y4,
34 X5, Y5, X6, Y6,
35 X7, Y7, X8, Y8);
36
37 // Add the polyline to a Pane.
38 Pane pane = new Pane(polyline);
39
40 // Create a Scene and display it.
41 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
42 primaryStage.setScene(scene);
43 primaryStage.show();
44 }
```

# The Text Class

The `Text` class draws a string on the screen as a graphic. Because the `Text` class is a subclass of the `Shape` class, it allows you to change its fill color, stroke, and other characteristics, just like any of the other shapes. The `Text` class has three constructors, summarized in [Table 13-10](#).

## Table 13-10 The Text class constructors

| Constructor                   | Description                                                                                                                                                                                                                                                                     |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Text()</code>           | Creates an empty <code>Text</code> object. Call the <code>Text</code> class's <code>setX</code> and <code>setY</code> methods to establish the <code>Text</code> object's location, and the <code>setText</code> method to establish the string that the object should display. |
| <code>Text(x, y, text)</code> | The <code>x</code> and <code>y</code> arguments, which are <code>doubles</code> , are the <i>XY</i> coordinates of the object's bottom-left corner. The <code>text</code> argument is the string that the object will display.                                                  |
| <code>Text(text)</code>       | The <code>text</code> argument is the string that the object will display. Call the <code>Text</code> class's <code>setX</code> and <code>setY</code> methods to establish the <code>Text</code> object's location.                                                             |

The following statement draws the string “Hello World”, starting at the coordinates 100, 50:

```
Text myText = new Text(100.0, 50.0, "Hello World");
```

You can set the font for the string with the `setFont` method. This method accepts a `Font` object as its argument. The `Font` class is in the `javafx.scene.text` package. When you instantiate the `Font` class, you pass

the name of a font and the font's size, in points, as arguments to the constructor. Here is an example that creates a `Text` object, sets the font to Serif 36 points, sets the stroke color to black, and sets the fill color to red:

```
myText.setFont(new Font("Serif", 36));
myText.setStroke(Color.BLACK);
myText.setFill(Color.RED);
```

[Code Listing 13-8](#) demonstrates the `Text` class. It draws the same octagon shape that the program in [Code Listing 13-5](#) class displayed, then draws the string “STOP” over it to create a stop sign. The string is drawn in a bold 38-point Sans Serif font. The program’s output is shown in [Figure 13-14](#).

## Figure 13-14 Output of StopSign.java



## Code Listing 13-8 (StopSign.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.paint.Color;
6 import javafx.scene.shape.Polygon;
7 import javafx.scene.text.Text;
8 import javafx.scene.text.Font;
9
```

```
10 public class StopSign extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Constants for the scene size
21 final double SCENE_WIDTH = 160.0;
22 final double SCENE_HEIGHT = 160.0;
23
24 // Constants for the vertices.
25 final double X1 = 60.0, Y1 = 20.0;
26 final double X2 = 100.0, Y2 = 20.0;
27 final double X3 = 140.0, Y3 = 60.0;
28 final double X4 = 140.0, Y4 = 100.0;
29 final double X5 = 100.0, Y5 = 140.0;
30 final double X6 = 60.0, Y6 = 140.0;
31 final double X7 = 20.0, Y7 = 100.0;
32 final double X8 = 20.0, Y8 = 60.0;
33
34 // Constants for the text
35 final double TEXT_X = 35.0;
36 final double TEXT_Y = 95.0;
37 final double FONT_SIZE = 38.0;
38
39 // Create an octagon.
40 Polygon octagon = new Polygon(X1, Y1, X2, Y2,
41 X3, Y3, X4, Y4,
42 X5, Y5, X6, Y6,
43 X7, Y7, X8, Y8);
44
45 // Set the octagon's fill color to red.
46 octagon.setFill(Color.RED);
47
48 // Create a Text object.
49 Text stopText = new Text(TEXT_X, TEXT_Y, "STOP");
50 stopText.setStroke(Color.WHITE);
51 stopText.setFill(Color.WHITE);
52 stopText.setFont(new Font("SansSerif", FONT_SIZE));
53
54 // Add the octagon to a Pane.
55 Pane pane = new Pane(octagon, stopText);
56
```

```
57 // Create a Scene and display it.
58 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
59 primaryStage.setScene(scene);
60 primaryStage.show();
61 }
62 }
```

## Rotating Nodes

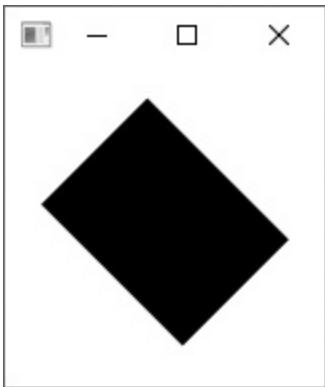
The `Node` class provides a method named `setRotate` that rotates a node about its center. Because the `setRotate` method is in the `Node` class, it can be used to rotate any node in your scene graph (including controls, such as `Buttons` and `Labels`).

When you call the method, you pass a `double` that is the angle of rotation, in degrees. The following code snippet is taken from the program `RotateBox.java` in the Student Sample Programs:

```
// Constants for the rectangle
final double X = 30.0, Y = 40.0;
final double WIDTH = 100.00, HEIGHT = 75.0;
final double ANGLE = 45.0;
// Create a rectangle.
Rectangle box = new Rectangle(X, Y, WIDTH, HEIGHT);
box.setRotate(ANGLE);
```

This code creates a rectangle, rotated at 45 degrees. The program's output is shown in [Figure 13-15](#).

## Figure 13-15 Output of RotateBox.java



If you rotate a container, the rotation will be applied to all children in the container. For example, [Code Listing 13-9](#) displays a rotated stop sign. This program is a modification of *StopSign.java*, that you saw in [Code Listing 13-7](#). In this version of the program, in line 57, the root node in the scene graph (a *Pane* object) is rotated. As a result, all of the objects in the scene graph are rotated 30 degrees. The program's output is shown in [Figure 13-16](#).

## Figure 13-16 Output of RotateStopSign.java



## Code Listing 13-9 (RotateStopSign.java)

```
1 import javafx.application.Application;
```

```
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.paint.Color;
6 import javafx.scene.shape.Polygon;
7 import javafx.scene.text.Text;
8 import javafx.scene.text.Font;
9
10 public class RotateStopSign extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Constants for the scene
21 final double SCENE_WIDTH = 160.0;
22 final double SCENE_HEIGHT = 160.0;
23 final double ANGLE = 30.0;
24
25 // Constants for the vertices
26 final double X1 = 60.0, Y1 = 20.0;
27 final double X2 = 100.0, Y2 = 20.0;
28 final double X3 = 140.0, Y3 = 60.0;
29 final double X4 = 140.0, Y4 = 100.0;
30 final double X5 = 100.0, Y5 = 140.0;
31 final double X6 = 60.0, Y6 = 140.0;
32 final double X7 = 20.0, Y7 = 100.0;
33 final double X8 = 20.0, Y8 = 60.0;
34
35 // Constants for the text
36 final double TEXT_X = 35.0;
37 final double TEXT_Y = 95.0;
38 final double FONT_SIZE = 38.0;
39
40 // Create an octagon.
41 Polygon octagon = new Polygon(X1, Y1, X2, Y2,
42 X3, Y3, X4, Y4,
43 X5, Y5, X6, Y6,
44 X7, Y7, X8, Y8);
45
46 // Set the octagon's fill color to red.
47 octagon.setFill(Color.RED);
48
```

```

49 // Create a Text object.
50 Text stopText = new Text(TEXT_X, TEXT_Y, "STOP");
51 stopText.setStroke(Color.WHITE);
52 stopText.setFill(Color.WHITE);
53 stopText.setFont(new Font("SansSerif", FONT_SIZE));
54
55 // Add the octagon to a Pane.
56 Pane pane = new Pane(octagon, stopText);
57 pane.setRotate(ANGLE);
58
59 // Create a Scene and display it.
60 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
61 primaryStage.setScene(scene);
62 primaryStage.show();
63 }
64 }
```

## Scaling Nodes

The `Node` class also provides methods named `setScaleX` and `setScaleY` that scale a node in its *X* and *Y* dimensions. Because these methods are in the `Node` class, they can be used to scale any node in your scene graph.

When you call the `setScaleX` and `setScaleY` methods, you pass a double that will be used as the scaling factor. For example, 2.0 will make the node twice its original size, and 0.5 will make the node half its original size. If you want the node to retain its original proportions, be sure to scale it the same in both the *X* and *Y* dimensions (in other words, pass the same value to the `setScaleX` and `setScaleY` methods). If you pass different values to these methods, the node will appear stretched in either the *X* or *Y* dimensions.

The following code snippet is taken from the program *ScaleText.java* in the Student Sample Programs:

```

// Constants for the text
final double X1 = 30.0, Y1 = 100.0;
final double X2 = 30.0, Y2 = 130.0;
final double X3 = 30.0, Y3 = 150.0;
final double FONT_SIZE = 38;
final double SCALE_HALF = 0.5;
final double SCALE_QTR = 0.25;
```

```
// Create three Text objects.
Text text1 = new Text(X1, Y1, "Hello World");
text1.setFont(new Font("SansSerif", FONT_SIZE));

Text text2 = new Text(X2, Y2, "Hello World");
text2.setFont(new Font("SansSerif", FONT_SIZE));
text2.setScaleX(SCALE_HALF);
text2.setScaleY(SCALE_HALF);

Text text3 = new Text(X3, Y3, "Hello World");
text3.setFont(new Font("SansSerif", FONT_SIZE));
text3.setScaleX(SCALE_QTR);
text3.setScaleY(SCALE_QTR);
```

This code creates three `Text` objects, each displaying the string “Hello World”. The first `Text` object is displayed at its original size. The second is scaled to half its original size, and the third is scaled to one-fourth its original size. The program’s output is shown in [Figure 13-17](#).

## Figure 13-17 Output of ScaleText.java



## Checkpoint

1. 13.1 What are the coordinates of the pixel in the upper-left corner of the

window?

2. 13.2 In a window 640 pixels wide by 480 pixels high, what are the coordinates of the pixel in the lower-right corner?
3. 13.3 How is the screen coordinate system different from the Cartesian coordinate system?
4. 13.4 In what package is the Shape class?
5. 13.5 In what package is the Color class?
6. 13.6 In what package is the Pane class?
7. 13.7 What shape method do you call to change the color of a shape's outline?
8. 13.8 What shape method do you call to change a shape's fill color?
9. 13.9 What shape method do you call to rotate a shape around its center?
10. 13.10 What shape method do you call to scale a shape along the X axis?
11. 13.11 What shape method do you call to scale a shape along the Y axis?
12. 13.12 Which of the following is not a subclass of the Shape class?
  1. Circle
  2. Rectangle
  3. Triangle
  4. Ellipse

## 13.2 Animation

### Concept:

JavaFX provides transition classes that create animations by causing a node to change, over time, from one state to another.

JavaFX provides a number of classes, known as *transition classes*, that make it easy to create animations. The transition classes are in the `javafx.animation` package, and some of them are listed in [Table 13-11](#).



VideoNote JavaFX Animation

**Table 13-11 Some of the transition classes**

| Class                            | Description                                                                                    |
|----------------------------------|------------------------------------------------------------------------------------------------|
| <code>TranslateTransition</code> | Creates an animation in which a node moves from one position to another over a period of time. |
| <code>RotateTransition</code>    | Creates an animation in which a node rotates over a period of time.                            |
| <code>ScaleTransition</code>     | Creates an animation in which a node becomes larger or smaller over a period of time.          |
| <code>StrokeTransition</code>    | Creates an animation in which a node's stroke color changes over a period of time.             |

|                             |                                                                                  |
|-----------------------------|----------------------------------------------------------------------------------|
| <code>FillTransition</code> | Creates an animation in which a node's fill color changes over a period of time. |
| <code>FadeTransition</code> | Creates an animation in which a node fades in or out over a period of time.      |

Although each of the transition classes creates a different type of animation, all of the classes work in a similar manner. For each class, you designate the following:

- The node that you want to animate
- The amount of time that the animation will last
- The node's starting state (depends on the type of animation)
- The node's ending state (depends on the type of animation)

Once you have everything properly setup, you call the transition class's `play` method to play the animation. Let's look at how this relates to each of the transition classes listed in [Table 13-10](#).

## The `TranslateTransition` Class

The `TranslateTransition` class causes a node to move from one position on the screen to another. The `TranslateTransition` class has three constructors, summarized in [Table 13-12](#).

**Table 13-12**  
**TranslateTransition class**  
**constructors**

| <b>Constructor</b> | <b>Description</b> |
|--------------------|--------------------|
|--------------------|--------------------|

`TranslateTransition()`

Creates an empty `TranslateTransition` object. Call the class's `setDuration` method to establish the duration of the animation, and the `setNode` method to specify the node to animate.

`TranslateTransition(duration)`

The `duration` argument, an object of the `Duration` class, specifies the amount of time that the animation should last. Call the `TranslateTransition` class's `setNode` method to specify the node to animate.

`TranslateTransition(duration, node)`

The `duration` argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The `node` argument, an object of the `Node` class (or its subclasses) is the node to animate.

You use a `Duration` object to specify the amount of time that an animation should last. The `Duration` class, which is in the `javafx.util` package, represents a duration of time. When you instantiate the `Duration` class, you pass a number of milliseconds as an argument to the constructor. A millisecond is 1/1000th of a second, so passing the value 1000 to the `Duration` class constructor will create an object that represents a duration of 1 second.

The following code snippet shows an example of creating a `Circle`, with its center point at (0, 50) and a radius of 30. Then, a `TranslateTransition` object is created to animate it. The duration of the animation is 3 seconds.

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
 new TranslateTransition(new Duration(3000), myCircle);
```

The next step is to specify the node's starting location. The `TranslateTransition` class has two methods named `setFromX` and `setFromY` for this purpose. You call these methods, passing the starting location's *X* and *Y* coordinates, respectively.

Next, you specify the node's ending location by calling the `TranslateTransition` class's `setToX` and `setToY` methods. You pass the ending location's *X* and *Y* coordinates, respectively, to these methods. Once you have everything setup, you call the transition class's `play` method to play the animation. The following code snippet shows an example:

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
 new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setToX(100);
ttrans.setToY(50);
ttrans.play();
```

This code creates a `Circle`, then creates a `TranslateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played, the `Circle` object will start at (0, 50), and it will move to the location (100, 50). It will take 3 seconds for the `Circle` object to complete the move.

Keep in mind that the values you pass to the `setToX` and `setToY` methods are the coordinates for the node's ending. In the previous code snippet, the circle's final position is (100, 50). If you prefer, you can call the `setByX` and `setByY` methods to specify the distance that the node should move, relative to its starting position. The following code snippet shows an example:

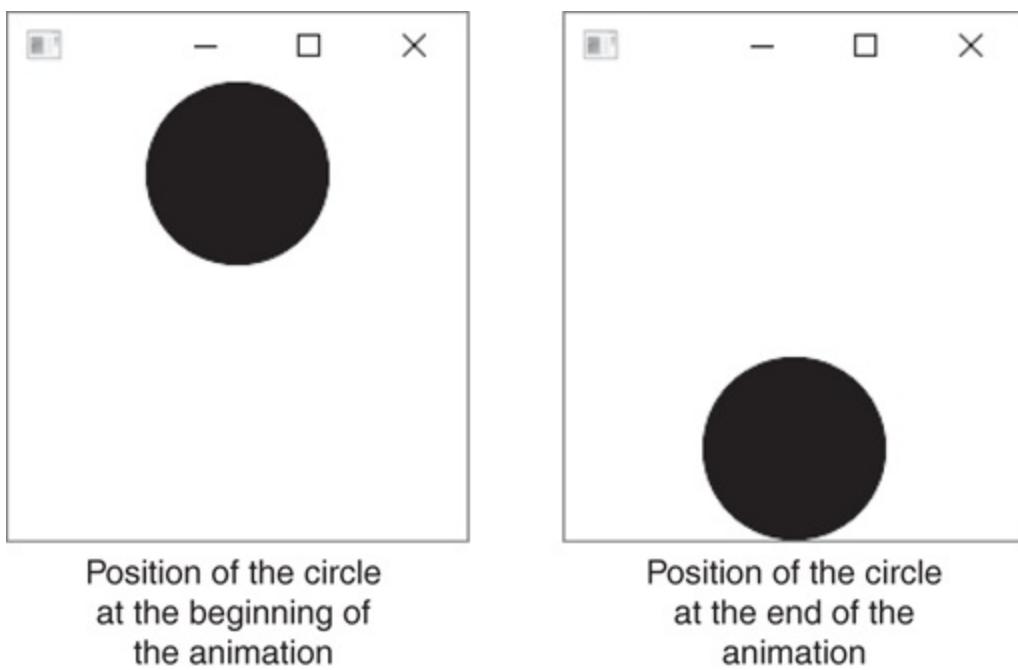
```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
 new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setByX(75);
ttrans.setByY(50);
ttrans.play();
```

In this example, the transition object initially positions the circle at (0, 50),

then moves it by an additional 75 pixels in the *X* direction, and an additional 50 pixels in the *Y* direction. So, the circle's ending position will be (75, 100).

[Code Listing 13-10](#) shows a complete program that animates a circle, moving it from the top of the application's window to the bottom. [Figure 13-18](#) shows the program's output.

## Figure 13-18 Output of BallDrop.java



## Code Listing 13-10 (BallDrop.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.shape.Circle;
5 import javafx.scene.layout.Pane;
6 import javafx.util.Duration;
7 import javafx.animation.TranslateTransition;
8
```

```
9 public class BallDrop extends Application
10 {
11 public static void main(String[] args)
12 {
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Constants for the scene size
20 final double SCENE_WIDTH = 200.0;
21 final double SCENE_HEIGHT = 200.0;
22
23 // Constants for the ball
24 final double START_X = 100.0, START_Y = 40.0;
25 final double END_X = 100.0, END_Y = 160.0;
26 final double RADIUS = 40.0;
27
28 // Constant for the animation duration
29 final double ONE_SEC = 1000.0;
30
31 // Create the ball.
32 Circle ball = new Circle(RADIUS);
33
34 // Create the Transition object.
35 TranslateTransition ttrans =
36 new TranslateTransition(new Duration(ONE_SEC), ball);
37 ttrans.setFromX(START_X);
38 ttrans.setFromY(START_Y);
39 ttrans.setToX(END_X);
40 ttrans.setToY(END_Y);
41
42 // Add the ball to a Pane.
43 Pane pane = new Pane(ball);
44
45 // Create a Scene and display it.
46 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
47 primaryStage.setScene(scene);
48 primaryStage.show();
49
50 // Play the animation.
51 ttrans.play();
52 }
53 }
```

# The RotateTransition Class

The `RotateTransition` class creates an animation in which a node rotates. The `RotateTransition` class has three constructors, summarized in [Table 13-13](#).

## Table 13-13 `RotateTransition` class constructors

| Constructor                                   | Description                                                                                                                                                                                                                                             |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>RotateTransition()</code>               | Creates an empty <code>RotateTransition</code> object. Call the class's <code>setDuration</code> method to establish the duration of the animation, and the <code>setNode</code> method to specify the node to animate.                                 |
| <code>RotateTransition(duration)</code>       | The <i>duration</i> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. Call the <code>RotateTransition</code> class's <code>setNode</code> method to specify the node to animate.     |
| <code>RotateTransition(duration, node)</code> | The <i>duration</i> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. The <i>node</i> argument, an object of the <code>Node</code> class (or its subclasses) is the node to animate. |

The following code snippet shows an example of creating a `Line`, then creating a `RotateTransition` object to animate it. The duration of the

animation is 5 seconds.

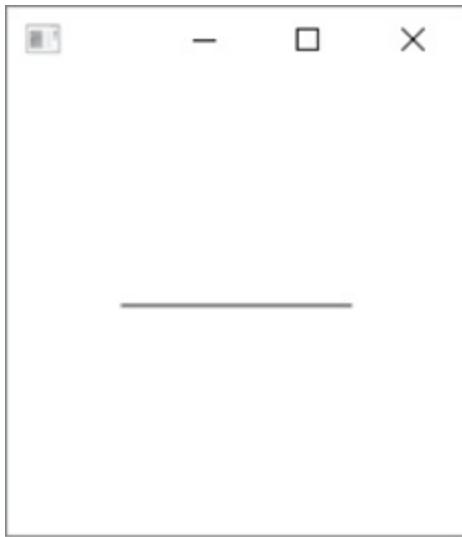
```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
 new RotateTransition(new Duration(5000), myLine);
```

The `RotateTransition` class rotates a node around its center point, from its starting angle, to its ending angle. You use the `RotateTransition` class's `setFromAngle` and `setToAngle` methods to specify the starting angle (the "from" angle) and the ending angle (the "to" angle). You pass an angle, in degrees, to each of these methods. The following code snippet shows an example:

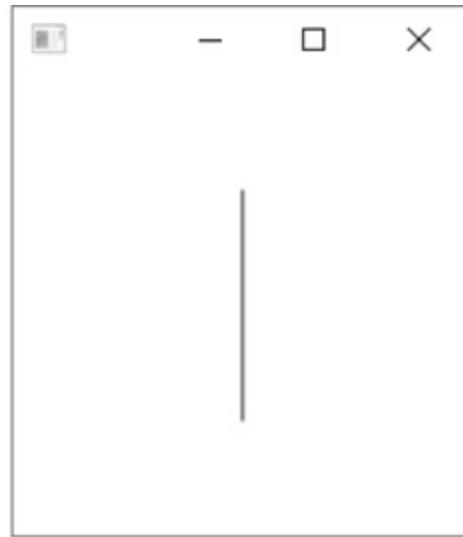
```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
 new RotateTransition(new Duration(3000), myLine);
rtrans.setFromAngle(0.0);
rtrans.setToAngle(90.0);
rtrans.play();
```

This code creates a `Line`, then creates a `RotateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played (by calling the `rtrans.play()` method), the `Line` object will be angled at 0 degrees, and it will rotate to the 90 degree position. It will take 3 seconds for the `Line` object to complete the rotation. [Figure 13-19](#) shows the position of the line at the beginning and end of the animation.

## Figure 13-19 A line rotating from 0 degrees to 90 degrees



Position of the line at the beginning of the animation



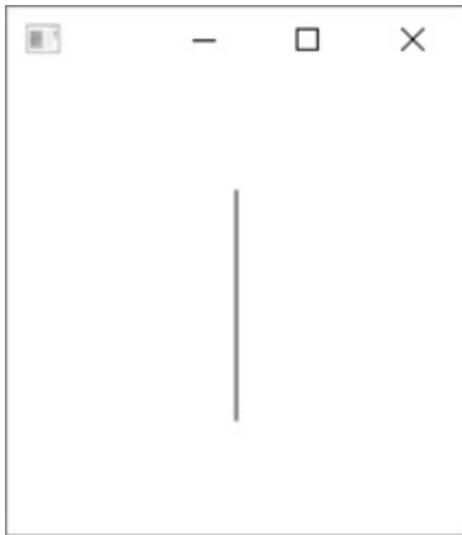
Position of the line at the end of the animation

Keep in mind the value you pass to the `setToAngle` method is the node's ending angle. In the previous code snippet, the line rotates from the 0 degree position to the 180 degree position. If you prefer, you can call the `setByAngle` method to specify the amount of rotation. The following code snippet shows an example:

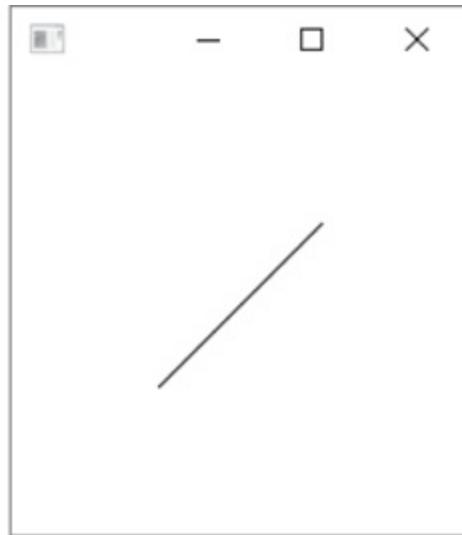
```
Line myLine = new Line(50, 100, 150, 100);
RotateTransition rtrans =
 new RotateTransition (new Duration(3000), myLine);
rtrans.setFromAngle(90.0);
rtrans.setByAngle(45.0);
rtrans.play();
```

In this example, the transition object initially positions the line at the 90 degree position, and rotates it by an additional 45 degrees. So, the ending angle will be 135 degrees. [Figure 13-20](#) shows the position of the line at the beginning and end of the animation.

## Figure 13-20 A line starting at 90 degrees, rotating by an additional 45 degrees



Position of the line at the beginning of the animation



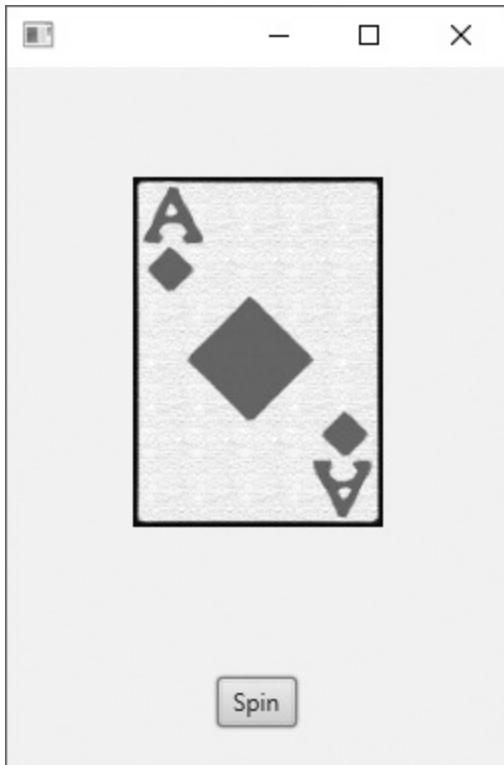
Position of the line at the end of the animation

## Tip:

If you pass a positive angle to the `setToAngle` or the `setByAngle` methods, the rotation will be clockwise. If you pass a negative angle to either of these methods, the rotation will be counterclockwise.

[Code Listing 13-11](#) shows a complete program that animates an image of a poker card. When the user clicks a button, the image spins around 360 degrees. [Figure 13-21](#) shows the program's output.

## Figure 13-21 Output of RotateImage.java



## Code Listing 13-11 (`RotateImage.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.BorderPane;
5 import javafx.scene.layout.HBox;
6 import javafx.scene.layout.VBox;
7 import javafx.geometry.Pos;
8 import javafx.geometry.Insets;
9 import javafx.scene.image.Image;
10 import javafx.scene.image.ImageView;
11 import javafx.animation.RotateTransition;
12 import javafx.util.Duration;
13 import javafx.scene.control.Button;
14 import javafx.event.EventHandler;
15 import javafx.event.ActionEvent;
16
17 /**
18 * An Animation Demo
19 */
```

```
20
21 public class RotateImage extends Application
22 {
23 public static void main(String[] args)
24 {
25 // Launch the application.
26 launch(args);
27 }
28
29 @Override
30 public void start(Stage primaryStage)
31 {
32 // Constants for the scene size
33 final double SCENE_WIDTH = 250.0;
34 final double SCENE_HEIGHT = 350.0;
35
36 // Constants for the animation
37 final double HALF_SEC = 500.0;
38 final double FROM_ANGLE = 0.0;
39 final double TO_ANGLE = 360.0;
40
41 // Constant for the layout padding space
42 final double PADDING_SPACE = 20.0;
43
44 // Create a BorderPane.
45 BorderPane borderPane = new BorderPane();
46
47 // Create an Image and ImageView components.
48 Image image = new Image("file:Ace_Diamonds.png");
49 ImageView imageView = new ImageView(image);
50
51 // Create a RotateTransition object for the ImageView.
52 RotateTransition rtrans =
53 new RotateTransition(new Duration(HALF_SEC), imageView);
54 rtrans.setFromAngle(FROM_ANGLE);
55 rtrans.setToAngle(TO_ANGLE);
56
57 // Put the ImageView in the center of the BorderPane.
58 borderPane.setCenter(imageView);
59
60 // Create a Button to start the animation.
61 Button spinButton = new Button("Spin");
62
63 // Create an event handler for the Button.
64 spinButton.setOnAction(event ->
65 {
66 rtrans.play();
67 });

```

```

68
69 // Put the Button in an HBox.
70 HBox hbox = new HBox(spinButton);
71 hbox.setAlignment(Pos.CENTER);
72 hbox.setPadding(new Insets(PADDING_SPACE)));
73
74 // Put the HBox in the bottom region of the BorderPane.
75 borderPane.setBottom(hbox);
76
77 // Create a Scene and display it.
78 Scene scene = new Scene(borderPane, SCENE_WIDTH, SCENE_HEIGHT);
79 primaryStage.setScene(scene);
80 primaryStage.show();
81 }
82 }
```

Let's take a closer look at the program.

- Lines 33 through 42 define various constants that are used in the program.
- Line 45 creates a `BorderPane` object, which will become the root node.
- Line 48 creates an `Image` component using the image of an Ace of Diamonds poker card, and line 49 creates an `ImageView` component to display the image.
- Lines 52 and 53 create a `RotateTransition` object for the animation. The duration of the animation is 500 milliseconds (half a second), and the `ImageView` component is the node that will be animated.
- Lines 54 and 55 set the starting angle to 0 degrees, and the ending angle to 360 degrees.
- Line 58 adds the `ImageView` component to the `BorderPane`'s center region.
- Line 61 creates a Button control named `spinButton`.
- Lines 64 through 67 use a lambda expression to register an event listener for the Button. The event listener plays the animation.

- Lines 70 through 72 put the `spinButton` control in an `HBox`, with center alignment and padding of 20 pixels.
- Line 75 adds the `HBox` to the `BorderPane`'s bottom region.
- Line 78 creates a scene with the `BorderPane` as the root node.
- Line 79 sets the scene to the stage, and line 80 displays the application's window.

## The `ScaleTransition` Class

The `ScaleTransition` class creates an animation in which a node becomes larger or smaller over time. The `ScaleTransition` class has three constructors, summarized in [Table 13-14](#).

**Table 13-14 `ScaleTransition` class constructors**

| <b>Constructor</b>                     | <b>Description</b>                                                                                                                                                                                                                                 |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ScaleTransition()</code>         | Creates an empty <code>ScaleTransition</code> object. Call the class's <code>setDuration</code> method to establish the duration of the animation, and the <code>setNode</code> method to specify the node to animate.                             |
| <code>ScaleTransition(duration)</code> | The <i>duration</i> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. Call the <code>ScaleTransition</code> class's <code>setNode</code> method to specify the node to animate. |

```
ScaleTransition(duration, node)
```

The *duration* argument, an object of the Duration class, specifies the amount of time that the animation should last. The *node* argument, an object of the Node class (or its subclasses) is the node to animate.

To animate a node with the ScaleTransition class, you specify starting scale factors along the *X* and *Y* axes, and ending scale factors along the *X* and *Y* axes. When the animation is played, the transition class will change the size of the node from the starting scale factors to the ending scale factors, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
ScaleTransition strans =
 new ScaleTransition(new Duration(2000), myCircle);
strans.setFromX(1.0);
strans.setFromY(1.0);
strans.setToX(3.0);
strans.setToY(3.0);
strans.play();
```

This code creates a Circle, with its center point at (50, 50) and a radius of 10. It then creates a ScaleTransition object to animate the Circle object. The duration of the animation is 2 seconds. When the animation is played, the Circle object will be displayed with a starting scale factor of 1 (along both the *X* and *Y* axes), and it will increase in size to a scale factor of 3 (along both the *X* and *Y* axes). It will take 2 seconds for the animation to complete.

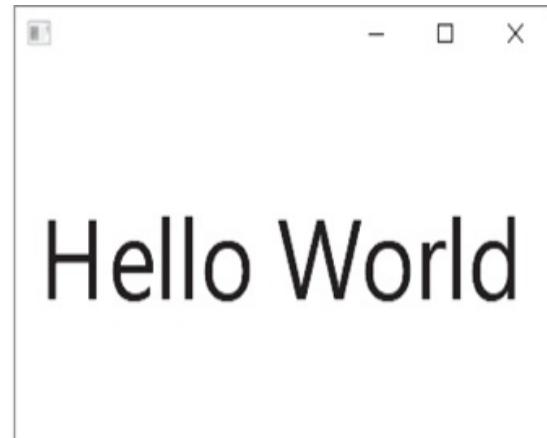
[Code Listing 13-12](#) shows a complete example. The program displays a Text object with a starting scale factor of 1 (along both axes) and over a duration of 5 seconds, it increases the Text object's size until it has an ending scale factor of 5. The program's output is shown in [Figure 13-22](#).

## Figure 13-22 An animation in which text is scaled to 5 times

# its original size



Size of the text at the beginning of the animation



Size of the text at the end of the animation

## Code Listing 13-12 (**ScaleTransitionText.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.geometry.Pos;
6 import javafx.scene.text.Text;
7 import javafx.util.Duration;
8 import javafx.animation.ScaleTransition;
9
10 public class ScaleTransitionText extends Application
11 {
12 public static void main(String[] args)
13 {
14 launch(args);
15 }
16
17 @Override
18 public void start(Stage primaryStage)
19 {
20 // Constants for the scene size
21 final double SCENE_WIDTH = 350.0;
22 final double SCENE_HEIGHT = 200.0;
```

```

23
24 // Constants for the transition
25 final double FIVE_SEC = 5000.0;
26 final double START_SCALE = 1.0;
27 final double END_SCALE = 5.0;
28
29 // Create a Text object.
30 Text text = new Text("Hello World");
31
32 // Create the Transition object.
33 ScaleTransition strans =
34 new ScaleTransition(new Duration(FIVE_SEC), text);
35 strans.setFromX(START_SCALE);
36 strans.setFromY(START_SCALE);
37 strans.setToX(END_SCALE);
38 strans.setToY(END_SCALE);
39
40 // Add the text to an HBox.
41 HBox hbox = new HBox(text);
42 hbox.setAlignment(Pos.CENTER);
43
44 // Create a Scene and display it.
45 Scene scene = new Scene(hbox, SCENE_WIDTH, SCENE_HEIGHT);
46 primaryStage.setScene(scene);
47 primaryStage.show();
48
49 // Play the animation.
50 strans.play();
51 }
52 }
```

Let's take a closer look at the program.

- Lines 21 through 27 define various constants.
- Line 30 creates a `Text` object that will display “Hello World”. Notice we do not specify an  $(X, Y)$  location for the text. This is okay, because we are going to put the `Text` object in an `HBox`. We will let the `HBox` position the text.
- Lines 33 and 34 create a `ScaleTransition` object to animate the `Text` object. The duration of the animation is 5 seconds.
- Line 35 sets the starting scale factor for the  $X$  axis to 1, and line 36 sets

the starting scale factor for the Y axis to 1.

- Line 37 sets the ending scale factor for the X axis to 5, and line 38 sets the ending scale factor for the Y axis to 5.
- Line 41 adds the `Text` object to an `HBox`, and line 42 sets the `HBox`'s alignment to center.
- Lines 45 through 47 create the scene, add the `HBox` to the scene as the root node, sets the scene to the stage, and displays the window.
- Line 50 plays the animation.

## The `StrokeTransition` Class

The `StrokeTransition` class creates an animation in which a shape's stroke color changes over a period of time. (Note the `StrokeTransition` class works only with objects of the `Shape` class, or one of its subclasses.) The `StrokeTransition` class has five constructors, summarized in [Table 13-15](#).

## Table 13-15 `StrokeTransition` class constructors

| Constructor                     | Description                                                                                                                                                                                                                                                                                                                                     |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>StrokeTransition()</code> | Creates an empty <code>StrokeTransition</code> object. Call the class's <code>setDuration</code> method to establish the duration of the animation, and the <code>setShape</code> method to specify the shape to animate, the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify |

|                                                |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                | <p>the ending color.</p> <p>The <i>duration</i> argument, an object of the Duration class, specifies the amount of time that the animation should last. Call the <i>StrokeTransition</i> class's <code>setShape</code> method to specify the shape to animate, the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify the ending color.</p> <p>The <i>duration</i> argument, an object of the Duration class, specifies the amount of time that the animation should last. The <i>shape</i> argument, an object of the Shape class (or its subclasses) is the node to animate. Call the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify the ending color.</p> <p>The <i>duration</i> argument, an object of the Duration class, specifies the amount of time that the animation should last. The <i>fromValue</i> argument, an object of the Color class, specifies the starting color. The <i>toValue</i> argument, an object of the Color class, specifies the ending color. Call the <code>setShape</code> method to specify the shape to animate.</p> <p>The <i>duration</i> argument, an object of the Duration class, specifies the amount of time that the animation</p> |
| <code>StrokeTransition(duration)</code>        |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>StrokeTransition(duration, shape)</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

should last. The *shape* argument, an object of the Shape class (or its subclasses) is the node to animate. The *fromValue* argument, an object of the Color class, specifies the starting color. The *toValue* argument, an object of the Color class, specifies the ending color. Call the `setShape` method to specify the shape to animate.

To animate a shape with the `StrokeTransition` class, you specify a starting color and an ending color. When the animation is played, the transition class will gradually change the shape's stroke color from the starting color to the ending color, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
StrokeTransition strtrans =
 new StrokeTransition(new Duration(2000), myCircle,
 Color.YELLOW, Color.GREEN);
strtrans.play();
```

This code creates a `Circle`, with its center point at (50, 50) and a radius of 10. It then creates a `StrokeTransition` object to animate the `Circle` object. The duration of the animation is 2 seconds. When the animation is played, the `Circle` object will be displayed with a starting stroke color of yellow, and it will gradually change the stroke color to green. It will take 2 seconds for the animation to complete.

## The `FillTransition` Class

The `FillTransition` class creates an animation in which a shape's fill color changes over a period of time. (Note the `FillTransition` class works only with objects of the `Shape` class, or one of its subclasses.) The `FillTransition` class has five constructors, summarized in [Table 13-16](#).

# Table 13-16 FillTransition class constructors

| Constructor                                  | Description                                                                                                                                                                                                                                                                                                                                                                                                        |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FillTransition()</code>                | Creates an empty <code>FillTransition</code> object. Call the class's <code>setDuration</code> method to establish the duration of the animation, and the <code>setShape</code> method to specify the shape to animate, the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify the ending color.                                                    |
| <code>FillTransition(duration)</code>        | The <code>duration</code> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. Call the <code>FillTransition</code> class's <code>setShape</code> method to specify the shape to animate, the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify the ending color.                  |
| <code>FillTransition(duration, shape)</code> | The <code>duration</code> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. The <code>shape</code> argument, an object of the <code>Shape</code> class (or its subclasses) is the node to animate. Call the <code>setFromValue</code> method to specify the starting color, and the <code>setToValue</code> method to specify the ending color. |
|                                              | The <code>duration</code> argument, an object of the <code>Duration</code> class, specifies the                                                                                                                                                                                                                                                                                                                    |

`FillTransition(duration, fromValue, toValue)`

amount of time that the animation should last. The *fromValue* argument, an object of the `Color` class, specifies the starting color. The *toValue* argument, an object of the `Color` class, specifies the ending color. Call the `setShape` method to specify the shape to animate.

`FillTransition(duration, shape, fromValue, toValue)`

The *duration* argument, an object of the `Duration` class, specifies the amount of time that the animation should last. The *shape* argument, an object of the `Shape` class (or its subclasses) is the node to animate. The *fromValue* argument, an object of the `Color` class, specifies the starting color. The *toValue* argument, an object of the `Color` class, specifies the ending color. Call the `setShape` method to specify the shape to animate.

To animate a shape with the `FillTransition` class, you specify a starting color and an ending color. When the animation is played, the transition class will gradually change the shape's fill color from the starting color to the ending color, over a specified duration of time. The following code snippet shows an example:

```
Circle myCircle = new Circle(50, 50, 10);
FillTransition ftrans =
 new FillTransition (new Duration(2000), myCircle,
 Color.LIGHTBLUE, Color.DARKBLUE);
ftrans.play();
```

This code creates a `Circle`, with its center point at (50, 50) and a radius of 10. It then creates a `FillTransition` object to animate the `Circle` object. The duration of the animation is 2 seconds. When the animation is played, the `circle` object will be displayed with a starting fill color of light blue, and it

will gradually change the fill color to dark blue. It will take 2 seconds for the animation to complete.

## The FadeTransition Class

The `FadeTransition` class creates an animation in which a node fades in or out over a period of time. The `FadeTransition` class has three constructors, summarized in [Table 13-17](#).

### Table 13-17 `FadeTransition` class constructors

| Constructor                                 | Description                                                                                                                                                                                                                                             |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>FadeTransition()</code>               | Creates an empty <code>FadeTransition</code> object. Call the class's <code>setDuration</code> method to establish the duration of the animation, and the <code>setNode</code> method to specify the node to animate.                                   |
| <code>FadeTransition(duration)</code>       | The <i>duration</i> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. Call the <code>ScaleTransition</code> class's <code>setNode</code> method to specify the node to animate.      |
| <code>FadeTransition(duration, node)</code> | The <i>duration</i> argument, an object of the <code>Duration</code> class, specifies the amount of time that the animation should last. The <i>node</i> argument, an object of the <code>Node</code> class (or its subclasses) is the node to animate. |

All nodes have an *opacity* value that determines whether the node is transparent or opaque. An opacity value is a double in the range of 0 through 1. When a node's opacity value is 0, the node is completely transparent. When a node's opacity value is 1, the node is completely opaque (you cannot see through it). When a node's opacity value is between 0 and 1, the node is semitransparent.

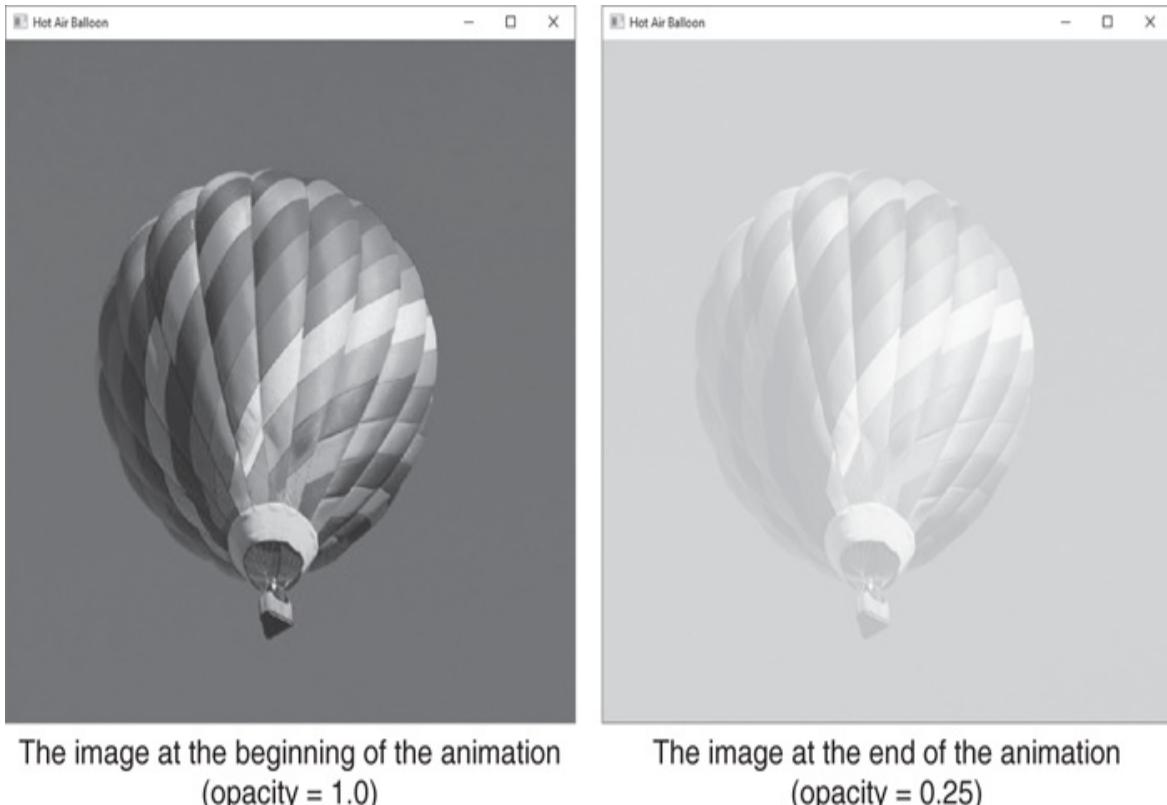
To animate a node with the `FadeTransition` class, you specify values for starting opacity and ending opacity. When the animation is played, the transition class will gradually change the node's opacity from the specified starting value to the specified ending value. The following code snippet shows an example:

```
// Create an Image and an ImageView.
Image image = new Image("file:HotAirBalloon.jpg");
ImageView imageView = new ImageView(image);

// Create a FadeTransition to animate the ImageView.
FadeTransition ftrans =
 new FadeTransition(new Duration(1000), imageView);
ftrans.setFromValue(1.0);
ftrans.setToValue(0.25);
ftrans.play();
```

This code creates an `Image` component and an `ImageView` component to display an image, then creates a `FadeTransition` object to animate the `ImageView` object. The duration of the animation is 1 second. When the animation is played, the image will be displayed with an opacity of 1.0 (it will be completely opaque). As the animation plays, the image's opacity will gradually decrease to 0.25 (twenty-five percent opaque). [Figure 13-23](#) shows an example of the code's output at the beginning and the end of the animation.

## Figure 13-23 Fade transition example



## Controlling the Animation

Each of the transition classes we have discussed are subclasses of the `Animation` class and the `Transition` class. [Table 13-18](#) summarizes a few of the methods inherited from the `Animation` class, that give you greater control over the animation.

### Table 13-18 A few methods inherited from the `Animation` class

#### Method

#### Description

The `time` argument is a `Duration` object.

|                                           |                                                                                                                                                                                                                                                                                                    |
|-------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>jumpTo(<i>time</i>)</code>          | This method causes the animation to jump to the time position specified by <i>time</i> . (For example, if the <i>time</i> argument specifies 5 seconds, the method will jump the animation to the 5-second position.)                                                                              |
| <code>pause()</code>                      | Pauses the animation. (This method has no effect if the animation is not playing.)                                                                                                                                                                                                                 |
| <code>play()</code>                       | Plays the animation from its current position. (This method has no effect if the animation is already running.)                                                                                                                                                                                    |
| <code>playFrom(<i>time</i>)</code>        | The <i>time</i> argument is a Duration object. This method plays the animation from the time position specified by <i>time</i> . (For example, if the animation lasts 6 seconds, and the <i>time</i> argument specifies 3 seconds, the method will play the animation from the 3-second position.) |
| <code>playFromStart()</code>              | Plays the animation from its beginning.                                                                                                                                                                                                                                                            |
| <code>setCycleCount(<i>value</i>)</code>  | The <i>value</i> argument is an integer. Defines the number of times the animation will repeat. If you want the animation to repeat indefinitely, specify the value <code>Animation.INDEFINITE</code> as the argument.                                                                             |
| <code>setAutoReverse(<i>value</i>)</code> | The <i>value</i> argument is a Boolean. If <i>value</i> is <code>true</code> , then the animation will play in reverse on every other cycle. If <i>value</i> is <code>false</code> , then the animation will play normally.                                                                        |
| <code>stop()</code>                       | Stops the animation, and resets the current position to the beginning of the animation.                                                                                                                                                                                                            |

# Specifying an Interpolator

All of the transition classes inherit a method named `setInterpolator` from the `Transition` class. The `setInterpolator` method allows you to specify how the animation begins and ends. The general format for the method is

```
transitionObject.setInterpolator(interpolatorValue);
```

where `transitionObject` is a transition object, and `interpolatorValue` is usually one of the constants listed in [Table 13-19](#). (The `Interpolator` class is in the `javafx.animation`. `Interpolator` package.)

## Table 13-19 Interpolator constants

| Method                              | Description                                                                                                                                                                             |
|-------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Interpolator.EASE_BOTH</code> | This interpolator causes the animation to ease in and ease out. In other words, the animation starts slowly, accelerates, then slows down at the end. This is the default interpolator. |
| <code>Interpolator.EASE_IN</code>   | This interpolator causes the animation to ease in (start slowly), but stop quickly.                                                                                                     |
| <code>Interpolator.EASE_OUT</code>  | This interpolator causes the animation to start quickly, but ease out (gradually slow down at the end).                                                                                 |
| <code>Interpolator.DISCRETE</code>  | With this interpolator, the animation appears to jump from its beginning immediately to its end.                                                                                        |
| <code>Interpolator.LINEAR</code>    | This interpolator causes the animated object to change its affected property (position, scale, rotation angle, color,                                                                   |

and so forth) at a constant rate over time.

The following code snippet shows an example:

```
Circle myCircle = new Circle(0, 50, 30);
TranslateTransition ttrans =
 new TranslateTransition(new Duration(3000), myCircle);
ttrans.setFromX(0);
ttrans.setFromY(50);
ttrans.setToX(100);
ttrans.setToY(50);
ttrans.setInterpolator(Interpolator.EASE_IN);
ttrans.play();
```

This code creates a `Circle`, then creates a `TranslateTransition` object to animate it. The duration of the animation is 3 seconds. When the animation is played, the `Circle` object will start at (0, 50), and it will move to the location (100, 50). It will take 3 seconds for the `Circle` object to complete the move. Because the interpolator is set to `Interpolator.EASE_IN`, the animation will start slowly, but end abruptly.



## Checkpoint

1. 13.13 Which transition class causes a node to move from one position on the screen to another?
2. 13.14 Which transition class causes a node to rotate?
3. 13.15 Which transition class causes a node to become larger or smaller?
4. 13.16 Which transition class causes a node's stroke color to change?
5. 13.17 Which transition class causes a node's fill color to change?
6. 13.18 Which transition class causes a node to fade?
7. 13.19 How do you specify the amount of time that an animation should

last?

8. 13.20 In what package is the Duration class?
9. 13.21 How can you change an animation's interpolator?

## 13.3 Effects

### Concept:

JavaFX provides several classes that can be used to apply special effects to nodes in the scene graph.

JavaFX provides several classes you can use to create special visual effects for the nodes in a scene graph. For example, you can give a node a 3D appearance by displaying a drop shadow or an inner shadow with it. You can also filter images by adding effects such as blurs and color adjustments.

[Table 13-20](#) lists several of the effect classes, each of which is in the `javafx.scene.effect` package.



VideoNote JavaFX Effects

### Table 13-20 Some of the effect classes

| Effect Class             | Description                                                                                                                                                   |
|--------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>DropShadow</code>  | Creates a drop shadow, which is a shadow that appears behind a node. You specify a color, radius, and offset for the drop shadow.                             |
| <code>InnerShadow</code> | Creates a shadow that appears inside the edges of a node, thus giving the node a recessed appearance. You specify a color, radius, and offset for the shadow. |

|              |                                                                                        |
|--------------|----------------------------------------------------------------------------------------|
| ColorAdjust  | Allows you to adjust an image's hue, saturation, brightness, and contrast.             |
| BoxBlur      | Creates a blurring effect using the box blur algorithm                                 |
| GaussianBlur | Creates a blurring effect using the Gaussian algorithm.                                |
| MotionBlur   | Blurs the pixels of an image in a specific direction, creating the illusion of motion. |
| SepiaTone    | Gives an image an antique, sepia tone effect                                           |
| Glow         | Creates a glowing effect                                                               |
| Reflection   | Creates a reflection of a node                                                         |

To create a special effect for a node, you create an object of the desired effect class, then call the node's `setEffect` method, passing the effect object as an argument. Let's look at how this relates to each of the effect classes listed in [Table 13-20](#).

## The DropShadow Class

Use the `DropShadow` class to create a shadow behind a node, giving it the appearance of being raised above a surface. [Code Listing 13-13](#) demonstrates how a drop shadow can be applied to a circle.

## Code Listing 13-13 (`DropShadowDemo.java`)

```

1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Circle;
6 import javafx.scene.paint.Color;
7 import javafx.scene.effect.DropShadow;
8

```

```

9 public class DropShadowDemo extends Application
10 {
11 public static void main(String[] args)
12 {
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Constants for the scene size
20 final double SCENE_WIDTH = 200.0;
21 final double SCENE_HEIGHT = 200.0;
22
23 // Constants for the Circle
24 final double X = 100.0, Y = 100.0, RADIUS = 50.0;
25
26 // Create a DropShadow
27 DropShadow dropShadow = new DropShadow();
28
29 // Create a Circle object.
30 Circle myCircle = new Circle(X, Y, RADIUS, Color.RED);
31 myCircle.setEffect(dropShadow);
32
33 // Add the Circle to a Pane.
34 Pane pane = new Pane(myCircle);
35
36 // Create a Scene and display it.
37 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
38 primaryStage.setScene(scene);
39 primaryStage.show();
40 }
41 }

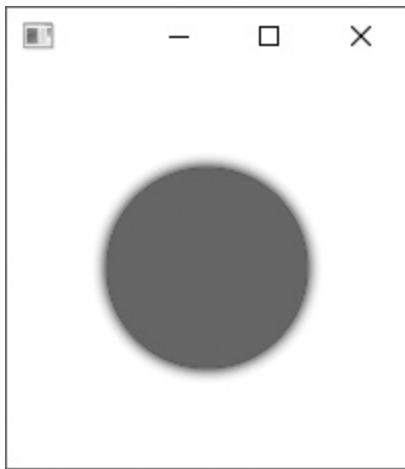
```

Here are the lines of code to take note of:

- Line 27 creates a `DropShadow` object named `dropShadow`, using the `DropShadow` class's no-arg constructor.
- Line 30 creates a `Circle` object.
- Line 31 calls the `circle` object's `setEffect` method, passing the `dropShadow` object as an argument.

The program's output is shown in [Figure 13-24](#).

# Figure 13-24 Output of DropShadowDemo.java



If you are not satisfied with the appearance of the default drop shadow, you can use the `DropShadow` class methods listed in [Table 13-21](#) to alter its appearance. For example, the following code snippet (taken from *DropShadowDemo2.java*) displays a circle with a drop shadow that has X and Y offsets of 10. As a result, the shadow is shifted 10 pixels to the right, and 10 pixels down. The code's output is shown in [Figure 13-25](#).

## Table 13-21 Some of the DropShadow class methods

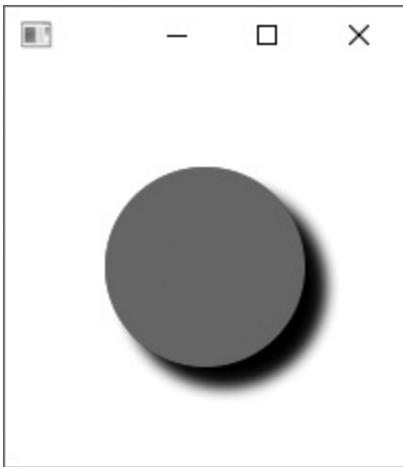
| Method                        | Description                                                                                                                         |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------------------------|
| <code>setColor(color)</code>  | The argument is a <code>Color</code> object that specifies the color of the shadow. The default value is <code>Color.BLACK</code> . |
| <code>setRadius(value)</code> | The argument is a <code>double</code> that specifies the radius of the shadow's blur. The default value is 10.0.                    |
|                               | The argument is a <code>double</code> between 0.0                                                                                   |

|                                           |                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setSpread(<i>value</i>)</code>      | and 1.0 that specifies how far the shadow spreads before it starts to blur. The default value is 0.0.                                                                                                                                                                                                                                                            |
| <code>setOffsetX(<i>value</i>)</code>     | The argument is a double that specifies the offset of the shadow in the X direction. The default value is 0.0.                                                                                                                                                                                                                                                   |
| <code>setOffsetY(<i>value</i>)</code>     | The argument is a double that specifies the offset of the shadow in the Y direction. The default value is 0.0.                                                                                                                                                                                                                                                   |
| <code>setBlurType(<i>blurType</i>)</code> | The argument is a <code>BlurType</code> enum constant (from the <code>javafx.scene.effect</code> package) that specifies the algorithm used to blur the shadow. The possible values are <code>BlurType.GAUSSIAN</code> , <code>BlurType.ONE_PASS_BOX</code> , <code>BlurType.TWO_PASS_BOX</code> , and <code>BlurType.THREE_PASS_BOX</code> (the default value). |

```
// Create a DropShadow
DropShadow dropShadow = new DropShadow();
dropShadow.setOffsetX(10);
dropShadow.setOffsetY(10);

// Create a Circle object.
Circle myCircle = new Circle(X, Y, RADIUS, Color.RED);
myCircle.setEffect(dropShadow);
```

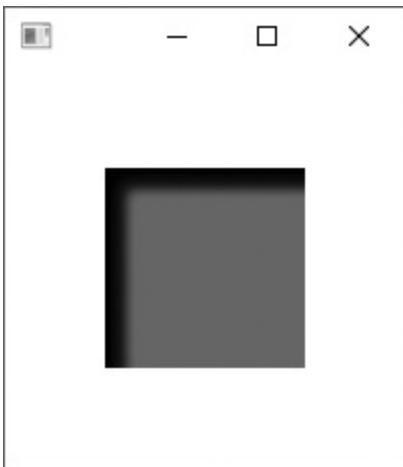
## Figure 13-25 Drop shadow with X and Y offsets of 10



## The InnerShadow Class

Use the `InnerShadow` class to create a shadow on the inside edge of a node, giving it a recessed appearance. The following code snippet, taken from `InnerShadowDemo.java`, demonstrates how an inner shadow can be applied to a rectangle. The code's output is shown in [Figure 13-26](#).

**Figure 13-26 A rectangle with an inner shadow**



```
// Create an InnerShadow
InnerShadow innerShadow = new InnerShadow();
```

```

innerShadow.setOffsetX(10);
innerShadow.setOffsetY(10);

// Create a Rectangle object.
Rectangle box = new Rectangle(X, Y, WIDTH, HEIGHT);
box.setFill(Color.RED);
box.setEffect(innerShadow);

```

Notice in the code snippet we call the `InnerShadow` class's `setOffsetX` and `setOffsetY` methods to shift the shadow. The `InnerShadow` class provides all of the methods that are listed for the `DropShadow` class in [Table 13-21](#) except the `setSpread` method. Instead, the `InnerShadow` class provides a method named `setChoke` that specifies how far the shadow spreads before it starts to blur. The default value is 0.0. The argument is a `double` between 0.0 and 1.0.

## The ColorAdjust Class

The `ColorAdjust` class allows you to adjust a node's hue, saturation, brightness, and contrast, at the pixel level. You can use the methods listed in [Table 13-22](#) to make these adjustments.

### Table 13-22 Some of the `ColorAdjust` class methods

| Method                     | Description                                                                                                                                                                                                                        |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setHue(value)</code> | This method adjusts the hue, or color, of each pixel in an image, shape, or other node. The argument, a <code>double</code> in the range of -1.0 through 1.0, represents the amount to change the color. The default value is 0.0. |
|                            | This method adjusts the saturation, or intensity, of each pixel in an image, shape, or other node. The argument, a <code>double</code> in                                                                                          |

`setSaturation(value)` the range of -1.0 through 1.0, represents the amount to change the saturation. The default value is 0.0.

`setBrightness(value)` This method adjusts the brightness of each pixel in an image, shape, or other node. The argument, a double in the range of -1.0 through 1.0, represents the amount to change the brightness. The default value is 0.0.

`setContrast(value)` This method adjusts the contrast, or the difference between bright areas and dark areas, in an image, shape, or other node. The argument, a double in the range of -1.0 through 1.0, represents the amount to change the color. The default value is 0.0.

The following code snippet shows an example of using each of these methods to manipulate an image:

```
ColorAdjust colorAdjust = new ColorAdjust();
colorAdjust.setHue(0.25);
colorAdjust.setSaturation(0.5);
colorAdjust.setBrightness(-0.25);
colorAdjust.setContrast(0.1);

Image image = new Image("file:flower.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(colorAdjust);
```

## The BoxBlur, GaussianBlur, and MotionBlur Classes

Each of these classes can be used to make a node look blurry, or out of focus. The `BoxBlur` class uses an algorithm in which a pixel's value is recalculated as the average of the values of neighboring pixels within an imaginary box. The `GaussianBlur` class tends to produce a smoother blur than the `BoxBlur`

class, but takes more processing time to create. The `MotionBlur` class blurs pixels in a specific direction, to create the illusion of motion. Let's look at each of these classes.

## The BoxBlur Class

The following code snippet shows an example of how to use the `BoxBlur` class to blur the text produced by a `Text` object. In [Figure 13-27](#), the text on the right is the text produced by this code snippet. The text on the left is the same text, without the blur effect.

### Figure 13-27 Comparison of normal text and text with a BoxBlur effect

Hello World

Hello World

```
// Make a BoxBlur.
BoxBlur boxBlur = new BoxBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(boxBlur);
```

You can increase the smoothness of the `BoxBlur` effect by increasing its number of iterations. By default, the class makes only one pass over the image. Each time it makes an additional pass, however, the blurring effect becomes smoother. You can call the class's `setIterations` method, passing an integer argument, to set the number of iterations. The maximum number of iterations is 3. Here is an example code snippet:

```
BoxBlur boxBlur = new BoxBlur();
boxBlur.setIterations(3);
```

If you want to increase or decrease the amount of blurriness produced by the `BoxBlur` class, you can use the `setWidth` and `setHeight` methods, passing a double argument, to change the size of the imaginary box that is used for recalculating each pixel's value. By default, the box is 5 pixels wide and 5 pixels high. The larger the box, the blurrier the effect. The minimum width and height is 1 pixel (which produces no effect at all) and the maximum width and height is 255 pixels. The following code snippet sets the width and height to 10 pixels. The code's output is shown in [Figure 13-28](#).

## Figure 13-28 `BlurBox` width and height set to 10 pixels

A screenshot of a Java application window titled "Java Application". Inside the window, the text "Hello World" is displayed twice. The text on the left is sharp and clear, while the text on the right is blurred, demonstrating the effect of the `BoxBlur` class.

```
// Make a BoxBlur.
BoxBlur boxBlur = new BoxBlur();
boxBlur.setWidth(10.0);
boxBlur.setHeight(10.0);

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(boxBlur);
```

## The GaussianBlur Class

The following code snippet shows an example of how to use the `GaussianBlur` class to blur the text produced by a `Text` object. In [Figure 13-29](#), the text on the right is the text that is produced by this code snippet. The text on the left is the same text, without the blur effect.

## Figure 13-29 Comparison of

# normal text and text with a GaussianBlur effect

Hello World

Hello World

```
// Make a GaussianBlur.
GaussianBlur gaussianBlur = new GaussianBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(gaussianBlur);
```

You can also increase or decrease the amount of blurriness produced by the `GaussianBlur` class by changing its blur radius. You call the `setRadius` method, passing the desired radius as a `double`. The larger the radius, the blurrier the effect. By default, the radius is 10.0. The minimum radius is 0.0 (which produces no effect at all) and the maximum radius is 63.0. Here is an example code snippet that sets the blur radius to 20.0:

```
GaussianBlur gaussianBlur = new GaussianBlur();
gaussianBlur.setRadius(20.0);
```

## The MotionBlur Class

The following code snippet shows an example of how to use the `MotionBlur` class to blur the text produced by a `Text` object. In [Figure 13-30](#), the text on the right is the text that is produced by this code snippet. The text on the left is the same text, without the blur effect.

## Figure 13-30 Comparison of normal text and text with a

# MotionBlur effect

Hello World      

```
// Make a MotionBlur.
MotionBlur motionBlur = new MotionBlur();

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(motionBlur);
```

The `MotionBlur` class blurs pixels in a particular direction. By default, the direction is 0.0 degrees, but you can change that by calling the class's `setAngle` method, passing the desired angle as a double. You can also increase or decrease the amount of blurriness produced by the `MotionBlur` class by changing its blur radius. You call the `setRadius` method, passing the desired radius as a double. The larger the radius, the blurrier the effect. By default, the radius is 10.0. The minimum radius is 0.0 (which produces no effect at all) and the maximum radius is 63.0.

The following code snippet creates a `MotionBlur` effect with a blur angle of 45 degrees, and a blur radius of 20.0. The effect is then applied to an `ImageView` object. [Figure 13-31](#) shows the code's output.

**Figure 13-31 Motion blur applied to an image**



```
// Make a MotionBlur.
MotionBlur motionBlur = new MotionBlur();
motionBlur.setAngle(45.0);
motionBlur.setRadius(20.0);

// Apply the effect to an image.
Image image = new Image("file:Porsche.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(motionBlur);
```

## The SepiaTone Class

You can use the `SepiaTone` class to give an image an antique, sepia tone appearance. The following code snippet applies the effect to an image:

```
// Make a SepiaTone effect.
SepiaTone sepia = new SepiaTone();

// Apply the effect to an image.
Image image = new Image("file:Ship.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(sepia);
```

You can change the intensity of the `SepiaTone` class's effect by calling the

`setLevel` method, passing a double in the range of 0.0 through 1.0. The higher the value, the more intense the effect. By default, the level is 1.0. Here is an example code snippet that sets the level to 0.5:

```
SepiaTone sepia = new SepiaTone();
sepia.setLevel(0.5);
```

## The Glow Class

You can use the `Glow` class to produce a glowing effect. The following code snippet applies the effect to an image:

```
// Make a Glow effect.
Glow glow = new Glow();

// Apply the effect to an image.
Image image = new Image("file:SanDiego.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(glow);
```

You can change the intensity of the `Glow` class's effect by calling the `setLevel` method, passing a double in the range of 0.0 through 1.0. The higher the value, the more intense the effect. By default, the level is 0.3. Here is an example code snippet that sets the level to 1.0:

```
Glow glow = new Glow();
glow.setLevel(0.5);
```

[Figure 13-32](#) shows how an image appears with and without the `Glow` effect. The image on the left is the original image, without the `Glow` effect, and the image on the right has a `Glow` effect applied, with the level set to 1.0.

## Figure 13-32 An image without, and with, the Glow effect



Image without the Glow effect

Image with the Glow effect

## The Reflection Class

You can use the `Reflection` class to create a reflection of a node. The reflection appears below the node. The following code snippet applies the effect to a `Text` object. The code's output is shown in [Figure 13-33](#).

### Figure 13-33 The Reflection effect applied to a Text object

Hello World  
Hello World

```
// Make a Reflection effect.
Reflection reflect = new Reflection();

// Create a Text object.
Text text = new Text("Hello World");
```

```
text.setFont(new Font("SansSerif", 36));
text.setEffect(reflect);
```

The `Reflection` class provides several methods, listed in [Table 13-23](#), for fine-tuning the way the reflection appears.

## Table 13-23 Some of the Reflection class methods

| Method                                      | Description                                                                                                                                                                                                                                       |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setTopOffset(<i>value</i>)</code>     | The <i>value</i> argument is a double. This method sets <i>value</i> as the distance from the bottom of the node and the top of the reflection. The default value is 0.0.                                                                         |
| <code>setFraction(<i>value</i>)</code>      | The <i>value</i> argument is a double. This method sets <i>value</i> as the fraction of the node that appears in the reflection. The default value is 0.75, which means that the bottom 75 percent of the node will appear in the reflection.     |
| <code>setTopOpacity(<i>value</i>)</code>    | The <i>value</i> argument is a double in the range of 0.0 through 1.0. This method sets <i>value</i> as the opacity of the reflection at its top edge, with 0.0 being fully transparent, and 1.0 being fully opaque. The default value is 0.5.    |
| <code>setBottomOpacity(<i>value</i>)</code> | The <i>value</i> argument is a double in the range of 0.0 through 1.0. This method sets <i>value</i> as the opacity of the reflection at its bottom edge, with 0.0 being fully transparent, and 1.0 being fully opaque. The default value is 0.0. |

# Combining Effects

The `Node` class's `setEffect` method accepts only one effect object as an argument. If you call the `setEffect` method more than once for a particular node, the last effect object that you pass as an argument replaces any previously passed effect objects for that node. However, it is possible to combine effects using the `Effect` class's `setInput` method. The following code snippet shows an example of combining a `DropShadow` with a `Reflection`, then applying them both to a `Text` object:

```
// Create a DropShadow effect
DropShadow dropShadow = new DropShadow();

// Create a Reflection effect.
Reflection reflect = new Reflection();

// Combine the Reflection with the DropShadow.
reflect.setInput(dropShadow);

// Create a Text object.
Text text = new Text("Hello World");
text.setFont(new Font("SansSerif", 36));
text.setEffect(reflect);
```

The `setInput` method “chains” effects together. When this code executes, the drop shadow effect is applied first, then the reflection effect is applied. The results are shown in [Figure 13-34](#). The following code snippet shows an example of chaining three effects together: a `DropShadow`, a `SepiaTone`, and a `Reflection`. The results are shown in [Figure 13-35](#). (The sepia tone effect is not noticeable when the image is printed in black and white, but you can try this code for yourself to see the true result.)

**Figure 13-34 Applying  
DropShadow and Reflection  
effects to a Text object**

Hello World

Hello World

**Figure 13-35 Applying DropShadow, SepiaTone, and Reflection effects to an image**



```
// Make a DropShadow effect.
DropShadow dropShadow = new DropShadow();

// Make a SepiaTone effect.
SepiaTone sepia = new SepiaTone();

// Make a Reflection effect.
Reflection reflect = new Reflection();

// Combine the DropShadow with the SepiaTone.
sepia.setInput(dropShadow);

// Combine the SepiaTone with the Reflection.
reflect.setInput(sepia);
```

```
// Apply the effects to an image.
Image image = new Image("file:Ship2.jpg");
ImageView imageView = new ImageView(image);
imageView.setEffect(reflect);
```



## Checkpoint

1. 13.22 In what package are the effect classes?
2. 13.23 Which effect class do you use to create a drop shadow?
3. 13.24 Which effect class do you use to create a shadow inside the edges of a node?
4. 13.25 Which effect class do you use to adjust an image's hue, saturation, brightness, and contrast?
5. 13.26 What three effect classes can you use to create blur effects?
6. 13.27 Which effect class do you use to create an antique sepia tone effect?
7. 13.28 Which effect class do you use to create a glowing effect?
8. 13.29 Which effect class do you use to create a reflection of a node?
9. 13.30 If you want to combine multiple effects on a single node, which Node class method would you use to set the effects?

# 13.4 Playing Sound Files

## Concept:

In JavaFX, you use the `Media` and `MediaPlayer` classes to play sound files. The `Media` class loads a sound file into memory, and the `MediaPlayer` class provides methods for playing the sound file.

You can use JavaFX to play audio that is stored in one of the popular file formats .AIF, .MP3, or .WAV. Playing a sound file in a JavaFX application is a two-step process: First, you load the sound file into memory, and second, you play the sound file. This requires two classes from the JavaFX library: `Media` and `MediaPlayer`. Both of these classes are in the `javafx.scene.media` package.



### Video Note Playing Sound Files with JavaFX

The `Media` class can load an audio file from the computer's local file system, or from an Internet location. To load an audio file, create an instance of the `Media` class, passing the constructor a *uniform resource locator*, or *URI*, that specifies the file's name and location. A simple way to get the URI for a file is to use the `File` class from the `java.io` package. The following code snippet demonstrates how we can use the `File` class and the `Media` class to load a sound file named *guitar.wav*:

```
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
```

In this code snippet, the first statement creates a `File` object named `soundFile`, representing the *guitar.wav* file. The second statement creates a `Media` object. Notice the argument we are passing to the `Media` class

constructor. The expression `soundFile.toURI().toString()` gives us a string version of the file's URI.

Because no path for the `guitar.wav` file was given in the first statement, it is assumed that the file is in the same directory or folder as the program's executable .class file. If you want to load a file from a different location, you can specify a path. Here is an example:

```
File soundFile = new File("C:\\media\\guitar.wav");
Media media = new Media(soundFile.toURI().toString());
```

Next, you create an instance of the `MediaPlayer` class, passing a reference to the `Media` object as an argument to the constructor. Here is an example:

```
MediaPlayer player = new MediaPlayer(media);
```

Once you have successfully created a `MediaPlayer` object, you can use the `MediaPlayer` methods listed in [Table 13-24](#) to control the way the sound file is played.

## Table 13-24 Some of the `MediaPlayer` methods

| Method                          | Description                                                                                                                                                                                                              |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>setAutoPlay(value)</code> | The argument is a boolean value. If you pass <code>true</code> to the method, the sound file will be immediately played once the <code>MediaPlayer</code> object is created and ready.                                   |
| <code>play()</code>             | Starts playing the sound file. Normally, this method plays the sound file from its beginning, but if the sound file is currently paused, the <code>play</code> method resumes playback at the point where it was paused. |
|                                 | Pauses the sound file. Subsequently                                                                                                                                                                                      |

|                                   |                                                                                                                                                                                                                                                      |
|-----------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>pause()</code>              | calling the <code>play</code> method will cause the sound file to resume playing at the point where it was paused.                                                                                                                                   |
| <code>stop()</code>               | Stops the playback of the sound file. Subsequently calling the <code>play</code> method will cause the sound file to play from its beginning.                                                                                                        |
| <code>setCycleCount(value)</code> | The argument is an <code>int</code> value that specifies the number of times that the sound file should be played. The default value is 1. If you pass the value <code>MediaPlayer.INDEFINITE</code> , the sound file will be repeated indefinitely. |

For example, the following code snippet loads the file `guitar.wav` and plays it as soon as the file is loaded and the `MediaPlayer` is ready:

```
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);
player.setAutoPlay(true);
```



## Note:

Neither the `Media` class nor the `MediaPlayer` class have any visual properties. Neither class is a subclass of `Node`, so you do not add objects of the `Media` class or the `MediaPlayer` class to your scene graph.

# Registering an EndOfMedia Event Handler

A `MediaPlayer` object has certain states that indicate what the `MediaPlayer` is currently doing. For example:

- When a sound file is loaded into memory and the `MediaPlayer` is ready to play it, the `MediaPlayer` is in the *READY* state.
- While the sound file is being played, the `MediaPlayer` is in the *PLAYING* state.
- While the sound file is paused, as a result of the `pause()` method, the `MediaPlayer` is in the *PAUSED* state.
- When the sound file has been stopped, as a result of the `stop()` method, the `MediaPlayer` is in the *STOPPED* state.

When a `MediaPlayer` object plays a sound file all the way to its end, the `MediaPlayer` object does not automatically transition to the *STOPPED* state. Instead, it remains in the *PLAYING* state. This can cause a problem in some applications, because the `play()` method will not play the sound file if the `MediaPlayer` object is already in the *PLAYING* state.

For example, suppose a program has a *play* button that should play a sound file every time the user clicks the button. The button's event handler calls the `MediaPlayer` object's `play()` method. The user clicks the *play* button and listens to the sound file all the way to its end. If the user clicks the *play* button again to replay the sound file, nothing happens because the `MediaPlayer` is already in the *PLAYING* state. In this type of application, you need a way to force the `MediaPlayer` object into the *STOPPED* state when it reaches the end of the sound file.

As it turns out, the `MediaPlayer` object fires an event when it reaches the end of the sound file. The event is known as the `EndOfMedia` event, and you can use the `setOnEndOfMedia` method to register an event handler that executes any time the event fires. In the event handler, you simply call the `MediaPlayer`'s `stop()` method to force the `MediaPlayer` into the *STOPPED* state. The following code snippet uses an anonymous inner class to create such an event handler:

```
// Load the sound file.
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);
```

```
// Register an OnEndOfMedia event handler.
player.setOnEndOfMedia(new Runnable()
{
 @Override
 public void run()
 {
 player.stop();
 }
});
```

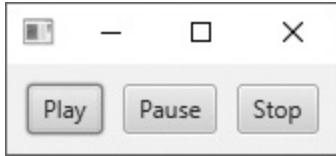
The event handler for the `EndOfMedia` event must be a class that implements the `Runnable` interface. (The `Runnable` interface is in the `java.lang` package, so it does not require an `import` statement.) The `Runnable` interface is a functional interface that specifies one method, named `run()`. Because `Runnable` is a functional interface, we can simplify the event handling code by writing it as a lambda expression. Here is an example:

```
// Load the sound file.
File soundFile = new File("guitar.wav");
Media media = new Media(soundFile.toURI().toString());
MediaPlayer player = new MediaPlayer(media);

// Register an OnEndOfMedia event handler.
player.setOnEndOfMedia(() ->
{
 player.stop();
});
```

[Code Listing 13-14](#) shows a complete example. The program loads the `guitar.wav` sound file, and it displays a play button, a pause button, and a stop button. These buttons allow the user to play, pause, and stop the sound file. The program's window is shown in [Figure 13-36](#).

## Figure 13-36 Window displayed by SoundPlayer.java



## Code Listing 13-14 (SoundPlayer.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.geometry.Pos;
6 import javafx.geometry.Insets;
7 import javafx.scene.media.Media;
8 import javafx.scene.media.MediaPlayer;
9 import javafx.scene.control.Button;
10 import java.io.File;
11
12 public class SoundPlayer extends Application
13 {
14 public static void main(String[] args)
15 {
16 // Launch the application.
17 launch(args);
18 }
19
20 @Override
21 public void start(Stage primaryStage)
22 {
23 // Load the sound file and create the MediaPlayer.
24 File soundFile = new File("guitar.wav");
25 Media media = new Media(soundFile.toURI().toString());
26 MediaPlayer player = new MediaPlayer(media);
27
28 // Event handler for the OnEndOfMedia event
29 player.setOnEndOfMedia(() ->
30 {
31 player.stop();
32 });
33
34 // Create the buttons.
35 Button playButton = new Button("Play");
```

```

36 Button pauseButton = new Button("Pause");
37 Button stopButton = new Button("Stop");
38
39 // Event handler for the play Button
40 playButton.setOnAction(event ->
41 {
42 player.play();
43 });
44
45 // Event handler for the pause Button
46 pauseButton.setOnAction(event ->
47 {
48 player.pause();
49 });
50
51 // Event handler for the stop Button
52 stopButton.setOnAction(event ->
53 {
54 player.stop();
55 });
56
57 // Put the Buttons in an HBox.
58 HBox hbox = new HBox(10, playButton, pauseButton, stopButton);
59 hbox.setAlignment(Pos.CENTER);
60 hbox.setPadding(new Insets(10));
61
62 // Create a Scene with the HBox as its root node.
63 Scene scene = new Scene(hbox);
64
65 // Add the Scene to the Stage and display the window.
66 primaryStage.setScene(scene);
67 primaryStage.show();
68 }
69 }
```



## Checkpoint

1. 13.31 What two classes do you use to play an audio file?
2. 13.32 Refer to your answer to [Checkpoint 13.31](#). In what package are the two classes?
3. 13.33 What does the MediaPlayer method setAutoPlay do?

4. 13.34 What does the MediaPlayer method setCycleCount do?

# 13.5 Playing Videos

## Concept:

In JavaFX, you use the `Media`, `MediaPlayer`, and `MediaView` classes to play videos. The `Media` class loads a video file into memory, and the `MediaPlayer` class provides methods for playing the video. The `MediaView` class can be inserted into the scene graph, providing a way to see the video on the screen.

Playing a video from a JavaFX application is similar to playing a sound file. First, you use the `Media` class to load the video (JavaFX supports the FLV and MP4 video formats), then you use the `MediaPlayer` class to control the video's playback. Because videos must be viewed on the screen, however, a third class named `MediaView` is required. (The `MediaView` class is in the `javafx.scene.media` package.) You simply create an instance of the `MediaView` class, passing the `MediaPlayer` object as an argument to the constructor.

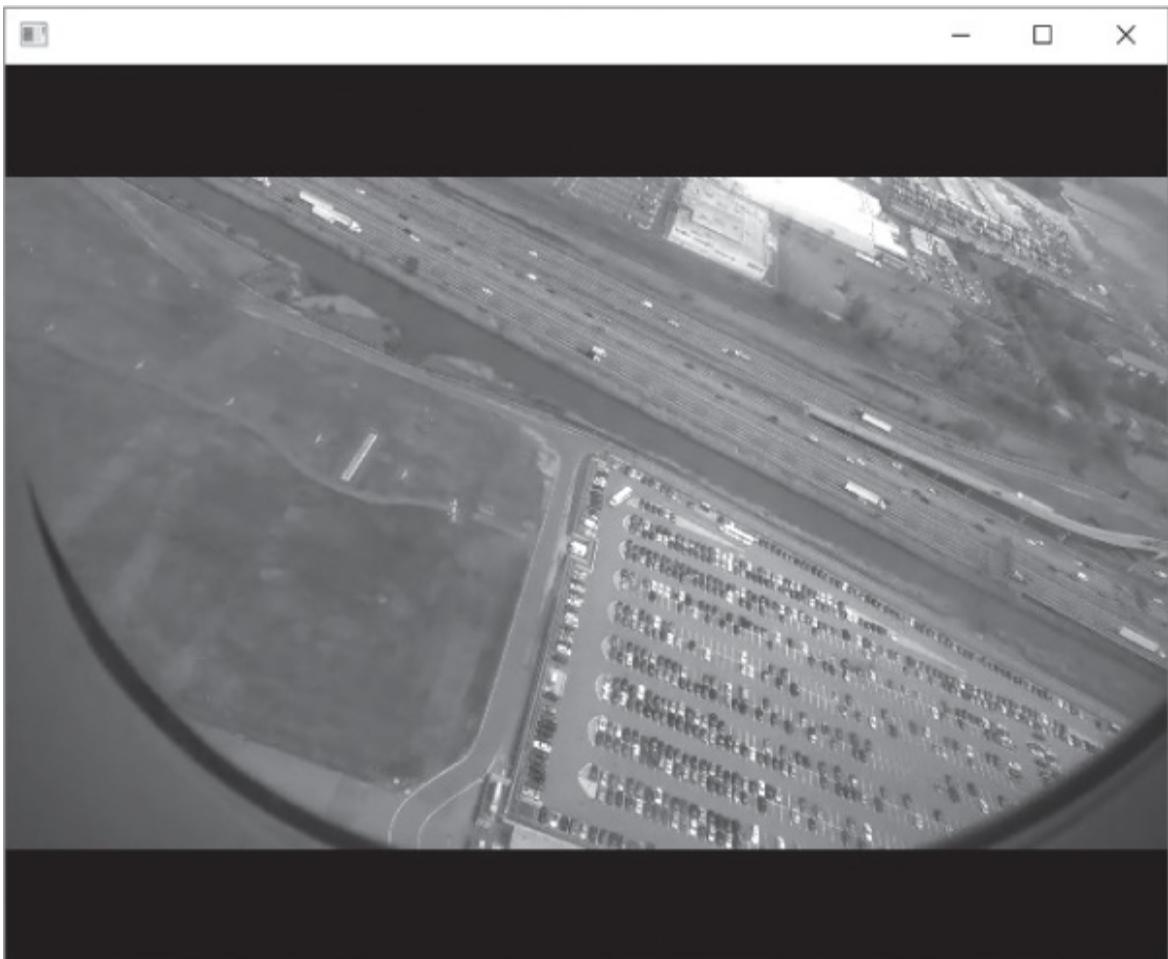


### VideoNote Playing Videos with JavaFX

The `MediaView` class is a subclass of `Node`, and you can add it to your scene graph. You set the size of the viewing window with the `setFitWidth` and `setFitHeight` methods. The video will be resized as necessary to fit within the viewing window. The program in [Code Listing 13-15](#) shows an example. When you run this program, it loads a video named *TakeOff.mp4* and automatically plays it. An example of the program's output is shown in [Figure 13-37](#).

## Figure 13-37 The

# **VideoDemo.java application**



## **Code Listing 13-15 (VideoDemo.java)**

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import java.io.File;
6 import javafx.scene.media.Media;
7 import javafx.scene.media.MediaPlayer;
8 import javafx.scene.media.MediaView;
9
```

```

10 public class VideoDemo extends Application
11 {
12 public static void main(String[] args)
13 {
14 // Launch the application.
15 launch(args);
16 }
17
18 @Override
19 public void start(Stage primaryStage)
20 {
21 // Constants for the video dimensions
22 final double WIDTH = 640.0, HEIGHT = 480.0;
23
24 // Load the video file.
25 File videoFile = new File("TakeOff.mp4");
26 Media media = new Media(videoFile.toURI().toString());
27
28 // Create the MediaPlayer and set AutoPlay to true.
29 MediaPlayer player = new MediaPlayer(media);
30 player.setAutoPlay(true);
31
32 // Create the MediaView.
33 MediaView view = new MediaView(player);
34
35 // Set the viewing dimensions.
36 view.setFitWidth(WIDTH);
37 view.setFitHeight(HEIGHT);
38
39 // Put the MediaView in an HBox.
40 HBox hbox = new HBox(view);
41
42 // Create a Scene, put it on the stage, and display it.
43 Scene scene = new Scene(hbox);
44 primaryStage.setScene(scene);
45 primaryStage.show();
46 }
47 }

```

Let's take a closer look at the program:

- Lines 25 and 26 load a file named *TakeOff.mp4*, and create a **Media** object for the file.
- Line 29 creates a **MediaPlayer** object, and line 30 sets the object's auto-play property to true. This causes the **MediaPlayer** object to

automatically play the video once everything is set up.

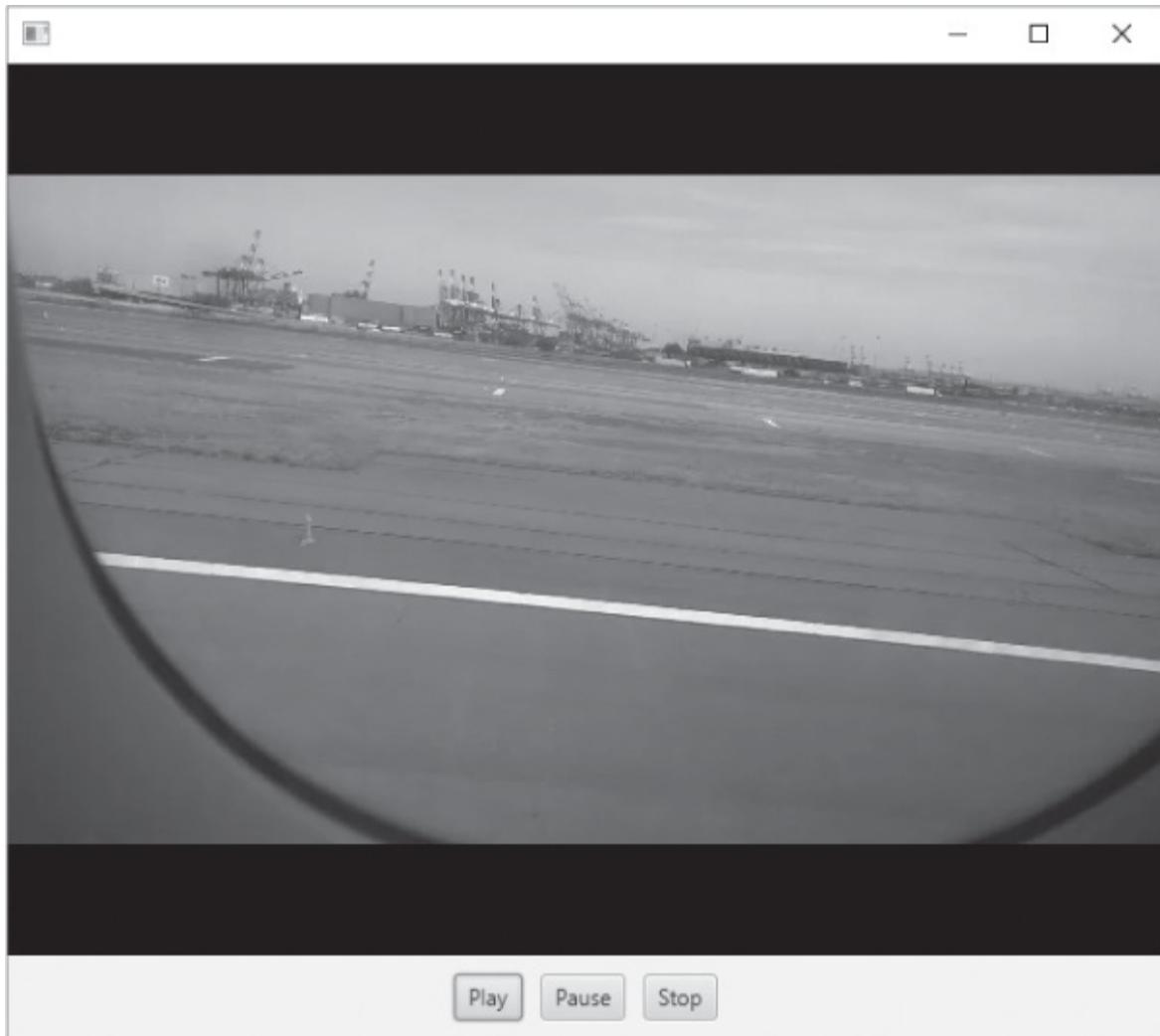
- Line 33 creates a `MediaView` object.
- Lines 36 and 37 set the `MediaView` object's width and height to 640 by 480.
- Line 40 puts the `MediaView` object in an `HBox`.
- Lines 43 through 45 create the scene graph with the `HBox` as its root node, sets the scene to the stage, and displays the application's window.

In the section on *Playing Sound Files*, we discussed an issue with the `MediaPlayer` that sometimes has to be dealt with. When a `MediaPlayer` object plays a media file (sound or video) all the way to its end, the `MediaPlayer` object does not automatically transition to the *STOPPED* state. Instead, it remains in the *PLAYING* state. This can cause a problem in some applications because the `play()` method will not play the media file if the `MediaPlayer` object is already in the *PLAYING* state.

So, if you want the ability to replay a video by calling the `play()` method after the video has already played to its end, you have to force the `MediaPlayer` object into the *STOPPED* state. You can accomplish this by writing an event handler for the `EndOfMedia` event. In the event handler, you simply call the `MediaPlayer`'s `stop()` method.

[Code Listing 13-16](#) shows a complete example. The program loads the *TakeOff.mp4* file, and it displays a *Play* button, a *Pause* button, and a *Stop* button. These buttons allow the user to play, pause, and stop the video. Notice in lines 36 through 39 we create an event handler for the `MediaPlayer` object's `EndOfMedia` event. Each time the video plays to its end, the event handler calls the `stop()` method, which forces the `MediaPlayer` into the *STOPPED* state. The program's window is shown in [Figure 13-38](#).

## Figure 13-38 The VideoPlayer.java application



## Code Listing 13-16 (VideoPlayer.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.layout.BorderPane;
6 import javafx.geometry.Pos;
7 import javafx.geometry.Insets;
8 import java.io.File;
9 import javafx.scene.media.Media;
10 import javafx.scene.media.MediaPlayer;
11 import javafx.scene.media.MediaView;
```

```
12 import javafx.scene.control.Button;
13
14 public class VideoPlayer extends Application
15 {
16 public static void main(String[] args)
17 {
18 // Launch the application.
19 launch(args);
20 }
21
22 @Override
23 public void start(Stage primaryStage)
24 {
25 // Constants for the video dimensions
26 final double WIDTH = 640.0, HEIGHT = 480.0;
27
28 // Load the video file.
29 File videoFile = new File("TakeOff.mp4");
30 Media media = new Media(videoFile.toURI().toString());
31
32 // Create the MediaPlayer.
33 MediaPlayer player = new MediaPlayer(media);
34
35 // Event handler for the OnEndOfMedia event
36 player.setOnEndOfMedia(() ->
37 {
38 player.stop();
39 });
40
41 // Create the MediaView.
42 MediaView view = new MediaView(player);
43
44 // Set the viewing dimensions.
45 view.setFitWidth(WIDTH);
46 view.setFitHeight(HEIGHT);
47
48 // Create the buttons.
49 Button playButton = new Button("Play");
50 Button pauseButton = new Button("Pause");
51 Button stopButton = new Button("Stop");
52
53 // Event handler for the play Button
54 playButton.setOnAction(event ->
55 {
56 player.play();
57 });
58
59 // Event handler for the pause Button
```

```

60 pauseButton.setOnAction(event ->
61 {
62 player.pause();
63 });
64
65 // Event handler for the stop Button
66 stopButton.setOnAction(event ->
67 {
68 player.stop();
69 });
70
71 // Put the Buttons in an HBox.
72 HBox hbox = new HBox(10, playButton, pauseButton, stopButton);
73 hbox.setAlignment(Pos.CENTER);
74 hbox.setPadding(new Insets(10));
75
76 // Put everything in a BorderPane.
77 BorderPane borderPane = new BorderPane();
78 borderPane.setCenter(view);
79 borderPane.setBottom(hbox);
80
81 // Create a Scene, put it on the stage, and display it.
82 Scene scene = new Scene(borderPane);
83 primaryStage.setScene(scene);
84 primaryStage.show();
85 }
86 }
```



## Checkpoint

1. 13.35 What three classes do you use to play a video file?
2. 13.36 Refer to your answer to [Checkpoint 13.35](#). In what package are the three classes?
3. 13.37 How do you set the size of a video's viewing window?

# 13.6 Handling Key Events

## Concept:

JavaFX allows you to create event handlers that respond to events generated by the keyboard.

*Key events* are events that are generated by the keyboard. There are three general types of key events:



**VideoNote** Handling Key Events in JavaFX

- Any time the user presses a key, a KEY\_PRESSED event occurs.
- Any time the user releases a key, a KEY\_RELEASED event occurs.
- Any time the user presses and releases a key that produces a Unicode character, such as a letter of the alphabet, a numeric digit, or a punctuation symbol, a KEY\_TYPED event occurs. (Keys such as the arrow keys, the page up and page down keys, and function keys do not generate KEY\_TYPED events.)

When a key event occurs, an event object that is an instance of the KeyEvent class is created. KeyEvent is a subclass of the Event class, both of which are in the `javafx.event` package. Suppose we want to write an event handler class that can respond to key events. This is the general format of an event handler class that handles event objects of the KeyEvent type:

```
class KeyHandler implements EventHandler<KeyEvent>
{
 @Override
 void handle(KeyEvent event)
```

```
{
 // Write event handling code here.
}
}
```

Notice the first line of the class declaration ends with the clause `implements EventHandler <KeyEvent>`. This specifies that the class implements the `EventHandler` interface, and the type of event object that it will work with is `KeyEvent`. Next, notice the `handle` method has a `KeyEvent` parameter named `event`. When the `handle` method is called, a `KeyEvent` object will be passed to it as an argument. Any code you want to execute as the result of a key being pressed or released must be written in the `handle` method.

Once you have written the event handler class, the next step is to register it with the desired node. Any node in your scene graph can handle key events, as long as that node has the input focus. When a node has the *input focus*, it is ready to receive input. You can tell which node has the input focus, while a JavaFX application is running, because it usually has a blue “glow” around it. When you press the Tab key, the input focus moves from one node to the next.

In most cases, you will want to register your `KeyEvent` handlers with the `Scene` object. That way, your `KeyEvent` handlers will respond to key events, regardless of which node has the input focus. To register a key event handler, you will call one of the following methods on the `Scene` object (or any node in the scene graph):

- `setOnKeyPressed`. Use this method to register an event handler for `KEY_PRESSED` events.
- `setOnKeyReleased`. Use this method to register an event handler for `KEY_RELEASED` events.
- `setOnKeyTyped`. Use this method to register an event handler for `KEY_TYPED` events.

## Using an Anonymous Inner Class to

# Register a Key Event Handler to the Scene

You learned in [Chapter 11](#) that a common technique for writing event handlers is to use anonymous inner classes. The following code snippet shows what it looks like to use an anonymous inner class to register a *KEY\_PRESSED* event handler for the scene:

```
// Create a Scene.
Scene scene = new Scene(rootNode);

// Register a KEY_PRESSED handler for the scene.
scene.setOnKeyPressed(new EventHandler<KeyEvent>()
{

 @Override
 public void handle(KeyEvent event)
 {
 // Write event-handling code here.
 }
});
```

# Using a Lambda Expression to Register a Key Event Handler to the Scene

You also learned in [Chapter 11](#) that lambda expressions provide a simplified way of registering event handlers. The following code snippet shows what it looks like to use a lambda expression to register a *KEY\_PRESSED* event handler for the scene:

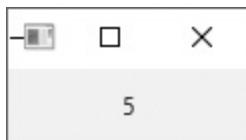
```
// Create a Scene.
Scene scene = new Scene(rootNode);

// Register a KEY_PRESSED handler for the scene.
```

```
scene.setOnKeyPressed(event ->
{
 // Write event-handling code here.
});
```

[Code Listing 13-17](#) shows a complete program that demonstrates how to handle *KEY\_PRESSED* events. The application simply increments a counter each time a key is pressed, and displays the value of the counter in a Label control. The application's window, after five key presses, is shown in [Figure 13-39](#).

## Figure 13-39 The KeyPressedDemo.java application after 5 keys have been pressed



## Code Listing 13-17 (KeyPressedDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.geometry.Pos;
6 import javafx.geometry.Insets;
7 import javafx.scene.control.Label;
8 import javafx.event.EventHandler;
9 import javafx.scene.input.KeyEvent;
10
11 public class KeyPressDemo extends Application
```

```

12 {
13 private int count = 0; // To keep count of key presses
14
15 public static void main(String[] args)
16 {
17 // Launch the application.
18 launch(args);
19 }
20
21 @Override
22 public void start(Stage primaryStage)
23 {
24 // Create a Label control.
25 Label label = new Label("0");
26
27 // Put the Label in an HBox.
28 HBox hbox = new HBox(10, label);
29 hbox.setAlignment(Pos.CENTER);
30 hbox.setPadding(new Insets(10));
31
32 // Create a Scene with the HBox as its root node.
33 Scene scene = new Scene(hbox);
34
35 // Register a KEY_PRESSED handler for the scene.
36 scene.setOnKeyPressed(event ->
37 {
38 count++;
39 label.setText(String.format("%d", count));
40 });
41
42 // Set the scene on the stage and display it.
43 primaryStage.setScene(scene);
44 primaryStage.show();
45 }
46 }
```

Let's take a closer look at the program:

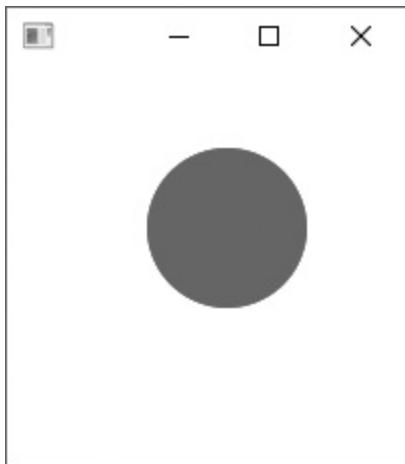
- Line 13 declares an `int` field named `count`, initialized with the value 0. The `count` field will keep count of the number of key presses.
- Line 25 creates a `Label` control that will display the number of key presses, and line 28 puts the `Label` control in an `HBox`.
- Line 33 creates a scene, with the `HBox` as the root node.

- Lines 36 through 40 use a lambda expression to register an event handler with the `Scene` object. The event handler will respond to `KEY_PRESSED` events. Inside the event handler, line 38 increments `count`, and line 39 displays the value of `count` in the `Label` control.
- Lines 43 and 44 set the scene to the stage and display the application's window.

The `KeyEvent` object that is passed as an argument to the event handler contains data about the key event. For example, you can use the object's `getCode()` method to determine which key was pressed to generate the event. The `getCode()` method returns a `KeyCode` value, which is an enum in the `javafx.scene.input` package. The `KeyCode` enum contains an extensive set of constants representing the keys on a computer keyboard. For example, `KeyCode.J` represents the J key, `KeyCode.ENTER` represents the Enter key, and `KeyCode.PAGE_UP` represents the Page Up key. See the JavaFX API documentation for the `KeyCode` enumeration for a list of all the constants.

[Code Listing 13-18](#) shows an example that lets the user move a circle around the screen using the arrow keys. An example of the program's screen is shown in [Figure 13-40](#).

## Figure 13-40 The MoveBall.java application



# Code Listing 13-18 (MoveBall.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.shape.Circle;
5 import javafx.scene.paint.Color;
6 import javafx.scene.layout.Pane;
7 import javafx.event.EventHandler;
8 import javafx.scene.input.KeyEvent;
9 import javafx.scene.input.KeyCode;
10
11 public class MoveBall extends Application
12 {
13 public static void main(String[] args)
14 {
15 launch(args);
16 }
17
18 @Override
19 public void start(Stage primaryStage)
20 {
21 // Constants for the scene size
22 final double SCENE_WIDTH = 200.0;
23 final double SCENE_HEIGHT = 200.0;
24
25 // Constants for the ball
26 final double START_X = 100.0, START_Y = 40.0;
27 final double RADIUS = 40.0;
28 final double STEP = 10.0;
29
30 // Create the ball.
31 Circle ball = new Circle(START_X, START_Y, RADIUS);
32 ball.setFill(Color.RED);
33
34 // Add the ball to a Pane.
35 Pane pane = new Pane(ball);
36
37 // Create a Scene and display it.
38 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
39
40 // Register a KEY_PRESSED handler for the scene.
41 scene.setOnKeyPressed(event ->
42 {
43 // Get the ball's coordinates.
```

```

44 double x = ball.getCenterX();
45 double y = ball.getCenterY();
46
47 // Check for the down-arrow key.
48 if (event.getCode() == KeyCode.DOWN &&
49 y < (SCENE_HEIGHT - RADIUS))
50 {
51 ball.setCenterY(y + STEP);
52 }
53
54 // Check for the up-arrow key.
55 if (event.getCode() == KeyCode.UP &&
56 y > RADIUS)
57 {
58 ball.setCenterY(y - STEP);
59 }
60
61 // Check for the right-arrow key.
62 if (event.getCode() == KeyCode.RIGHT &&
63 x < (SCENE_WIDTH - RADIUS))
64 {
65 ball.setCenterX(x + STEP);
66 }
67
68 // Check for the left-arrow key.
69 if (event.getCode() == KeyCode.LEFT &&
70 x > RADIUS)
71 {
72 ball.setCenterX(x - STEP);
73 }
74 });
75
76 // Set the scene on the stage and display it.
77 primaryStage.setScene(scene);
78 primaryStage.show();
79 }
80 }
```

Let's take a closer look at the program:

- Lines 22 and 23 define constants for the scene dimensions.
- Lines 26 through 28 define constants for the `circle` object.
- Line 31 creates the `Circle` object, and line 32 sets its color to red. Line 35 adds the `Circle` object to a `Pane`, and line 38 creates a `Scene`, with the

Pane as the root node.

- Lines 41 through 74 uses a lambda expression to register an event handler for the scene. The event handler will respond to *KEY\_PRESSED* events.
  - Lines 44 and 45 get the coordinates of the `circle` object's center point, and assign them to the variables `x` and `y`.
  - The `if` statement in lines 48 through 52 determines whether the key that was pressed was the down-arrow, and whether the bottom of the `circle` object is above the bottom edge of the window. If these conditions are true, line 51 moves the `circle` object down by 10 pixels.
  - The `if` statement in lines 55 through 59 determines whether the key that was pressed was the up-arrow, *and* whether the top of the `circle` object is below the top edge of the window. If these conditions are true, line 58 moves the `circle` object up by 10 pixels.
  - The `if` statement in lines 62 through 66 determines whether the key that was pressed was the right-arrow, *and* whether the right side of the `circle` object is not at the right edge of the window. If these conditions are true, line 65 moves the `circle` object to the right by 10 pixels.
  - The `if` statement in lines 69 through 73 determines whether the key that was pressed was the left-arrow, *and* whether the left side of the `circle` object is not at the left edge of the window. If these conditions are true, line 72 moves the `circle` object to the left by 10 pixels.
- Lines 77 and 78 set the scene to the stage and display the application's window.



## Checkpoint

1. 13.40 What type of event happens when the user presses a key on the keyboard?
2. 13.41 What type of event happens when the user releases a key on the keyboard?
3. 13.42 What type of event happens when the user presses and releases a key that produces a Unicode character?
4. 13.43 What KeyEvent method can you call to determine which key the user pressed?
5. 13.44 Refer to your answer for [Checkpoint 13.43](#). What does the method return?

# 13.7 Handling Mouse Events

## Concept:

JavaFX allows you to create event handlers that respond to events generated by the mouse.

The mouse generates several types of events, most of which are described in [Table 13-25](#). When a mouse event occurs, a `MouseEvent` object is created and delivered to the node that is under the mouse cursor. For example, if the user clicks the mouse button while the mouse cursor is over an `ImageView` control, a `MouseEvent` object will be delivered to the `ImageView` control. Likewise, if the user drags the mouse while the mouse cursor is over a `Circle` node, a `MouseEvent` object will be delivered to the `Circle` node.

## Table 13-25 Mouse Event Types

| Mouse Event Type           | Description                                                                      |
|----------------------------|----------------------------------------------------------------------------------|
| <code>MOUSE_CLICKED</code> | This event occurs when the user presses and releases the mouse button.           |
| <code>MOUSE_DRAGGED</code> | This event occurs when the user moves the mouse while pressing the mouse button. |
| <code>MOUSE_ENTERED</code> | This event occurs when the mouse cursor enters the space occupied by a node.     |
| <code>MOUSE_EXITED</code>  | This event occurs when the mouse cursor exits the space occupied by a node.      |

|                       |                                                                                   |
|-----------------------|-----------------------------------------------------------------------------------|
| <i>MOUSE_MOVED</i>    | This event occurs when the mouse is moved without the mouse button being pressed. |
| <i>MOUSE_PRESSED</i>  | This event occurs when the user presses the mouse button.                         |
| <i>MOUSE_RELEASED</i> | This event occurs when the user releases the mouse button.                        |



### VideoNote Handling Mouse Events in JavaFX

It is the responsibility of each node to handle (or ignore) any mouse events that are delivered to the node. For example, if your application displays a Rectangle, and you want an action to take place when the user clicks the Rectangle, then you must write an event handler that responds to **MOUSE\_CLICKED** events, and register that event handler with the Rectangle.

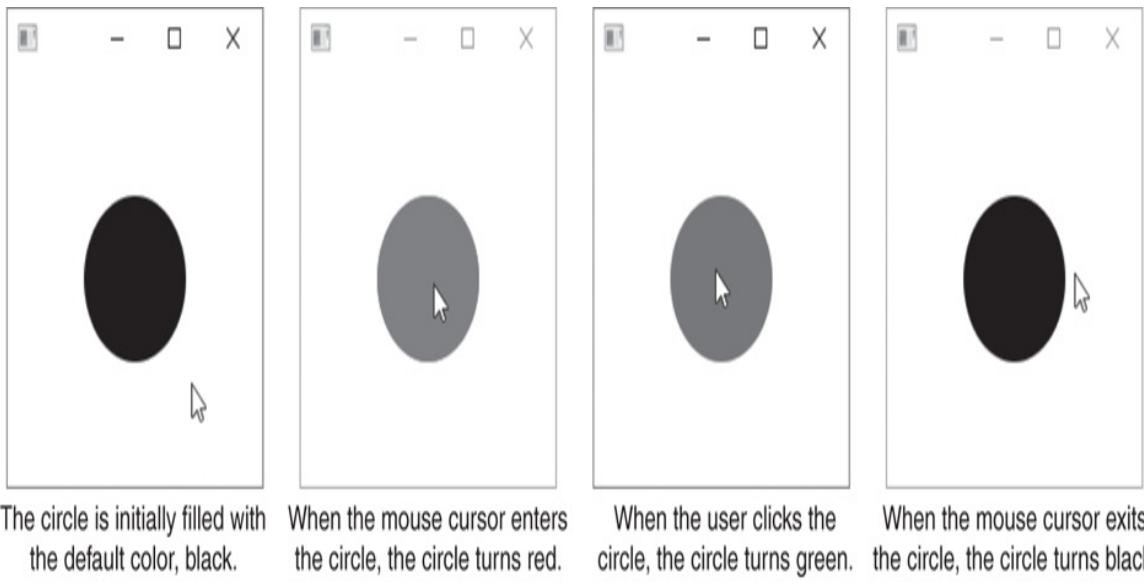
To register a key event handler, you will call one of the following methods on the node that should handle the event:

- **setOnMouseClicked.** Use this method to register an event handler for **MOUSE\_CLICKED** events.
- **setOnMouseDragged.** Use this method to register an event handler for **MOUSE\_DRAGGED** events.
- **setOnMouseEntered.** Use this method to register an event handler for **MOUSE\_ENTERED** events.
- **setOnMouseExited.** Use this method to register an event handler for **MOUSE\_EXITED** events.
- **setOnMouseMoved.** Use this method to register an event handler for **MOUSE\_MOVED** events.
- **setOnMousePressed.** Use this method to register an event handler for **MOUSE\_PRESSED** events.

- `setOnMouseReleased`. Use this method to register an event handler for `MOUSE_RELEASED` events.

[Code Listing 13-19](#) shows a complete program that demonstrates how to handle mouse events. The program draws a circle, filled with the default color of black, in the center of the window. Then, in lines 31 through 44, it registers event handlers for the `MOUSE_ENTERED`, `MOUSE_EXITED`, and `MOUSE_PRESSED` events. When the mouse cursor moves into the circle, the circle turns red. When the mouse cursor moves out of the circle, the circle turns back to black. When the user clicks the circle, the circle turns green. This is described in [Figure 13-41](#).

## Figure 13-41 The MouseEventDemo.java application



[Figure 13-41 Full Alternative Text](#)

## Code Listing 13-19

# (MouseEventDemo.java)

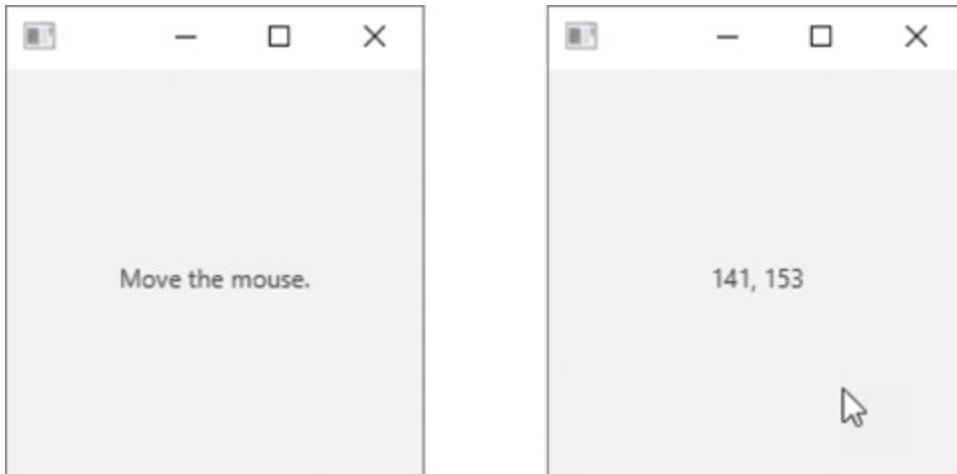
```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.shape.Circle;
5 import javafx.scene.paint.Color;
6 import javafx.scene.layout.Pane;
7
8 public class MouseEventDemo extends Application
9 {
10 public static void main(String[] args)
11 {
12 // Launch the application.
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Constants for the scene size
20 final double SCENE_WIDTH = 200.0;
21 final double SCENE_HEIGHT = 200.0;
22
23 // Constants for the ball
24 final double CENTER_X = 100.0, CENTER_Y = 100.0;
25 final double RADIUS = 40.0;
26
27 // Create the ball.
28 Circle ball = new Circle(CENTER_X, CENTER_Y, RADIUS);
29
30 // Register mouse event handlers with the ball.
31 ball.setOnMouseEntered(event ->
32 {
33 ball.setFill(Color.RED);
34 });
35
36 ball.setOnMouseExited(event ->
37 {
38 ball.setFill(Color.BLACK);
39 });
40
41 ball.setOnMousePressed(event ->
42 {
43 ball.setFill(Color.GREEN);
44 });

```

```
45
46 // Add the ball to a Pane.
47 Pane pane = new Pane(ball);
48
49 // Create a Scene and display it.
50 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
51 primaryStage.setScene(scene);
52 primaryStage.show();
53 }
54 }
```

In your mouse event handler, if you want to get the mouse cursor's location, you can call the `MouseEvent` object's `getSceneX` and `getSceneY` methods. These methods return the mouse cursor's *X* and *Y* coordinates, respectively. For example, look at [Code Listing 13-20](#). The program displays a window with a `Label` control that tells the user to "Move the mouse." When the user moves the mouse inside the application's window, the (*X*, *Y*) coordinates of the mouse cursor are continuously displayed in the `Label` control. The program's output is shown in [Figure 13-42](#).

## Figure 13-42 The MouseMovedDemo.java application



The initial message.

The mouse cursor's coordinates displayed.

## Code Listing 13-20 (**MouseMovedDemo.java**)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.HBox;
5 import javafx.scene.control.Label;
6 import javafx.geometry.Pos;
7
8 public class MouseMovedDemo extends Application
9 {
10 public static void main(String[] args)
11 {
12 // Launch the application.
13 launch(args);
14 }
15
16 @Override
17 public void start(Stage primaryStage)
18 {
19 // Constants for the scene size
20 final double SCENE_WIDTH = 200.0;
21 final double SCENE_HEIGHT = 200.0;
22
23 // Create a Label.
24 Label label = new Label("Move the mouse.");
```

```

25
26 // Add the Label to a Pane.
27 HBox hbox = new HBox(label);
28 hbox.setAlignment(Pos.CENTER);
29
30 // Create a Scene.
31 Scene scene = new Scene(hbox, SCENE_WIDTH, SCENE_HEIGHT);
32
33 // Register an event handler to the scene for
34 // the MOUSE_MOVED event.
35 scene.setOnMouseMoved(event ->
36 {
37 // Get the mouse cursor's coordinates.
38 double x = event.getSceneX();
39 double y = event.getSceneY();
40
41 // Display the mouse cursor's location.
42 label.setText(String.format("%.0f, %.0f", x, y));
43 });
44
45 // Set the scene to the stage and display it.
46 primaryStage.setScene(scene);
47 primaryStage.show();
48 }
49 }
```

Notice in [Code Listing 13-20](#) the mouse event handler (in lines 35 through 43) is registered to the Scene object, not the Label control. This is because we want to get mouse events as long as the mouse cursor is anywhere inside the application's window. If we had registered the event handler to the Label control, we would get mouse events only when the mouse cursor was over the Label control.

The *MOUSE\_DRAGGED* event fires when the user moves the mouse while pressing the mouse button. The event continues to fire while the user performs this action. [Code Listing 13-21](#) demonstrates this. The program draws a circle, initially positioned in the center of the window. It then registers an event handler that lets the user drag the circle with the mouse.

## Code Listing 13-21 (MouseDraggedDemo.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Circle;
6
7 public class MouseDraggedDemo extends Application
8 {
9 public static void main(String[] args)
10 {
11 // Launch the application.
12 launch(args);
13 }
14
15 @Override
16 public void start(Stage primaryStage)
17 {
18 // Constants for the scene size
19 final double SCENE_WIDTH = 200.0;
20 final double SCENE_HEIGHT = 200.0;
21
22 // Constants for the ball
23 final double CENTER_X = 100.0, CENTER_Y = 100.0;
24 final double RADIUS = 40.0;
25
26 // Create the ball.
27 Circle circle = new Circle(CENTER_X, CENTER_Y, RADIUS);
28
29 // Register an event handler for
30 // the MOUSE_DRAGGED event.
31 circle.setOnMouseDragged(event ->
32 {
33 // Get the mouse cursor's coordinates.
34 double x = event.getSceneX();
35 double y = event.getSceneY();
36
37 // Move the circle.
38 circle.setCenterX(x);
39 circle.setCenterY(y);
40 });
41
42 // Add the circle to a Pane.
43 Pane pane = new Pane(circle);
44
45 // Create a Scene and display it.
46 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
47 primaryStage.setScene(scene);
48 primaryStage.show();
```

```
49 }
50 }
```

Let's take a closer look at the mouse event handler, which is registered to the `circle` object in lines 31 through 40. Lines 34 and 35 get the mouse cursor's coordinates and assigns them to the `x` and `y` variables. Then, lines 38 and 39 move the circle's center point to these coordinates. As a result, the circle's center point is moved to the same location as the mouse cursor.

## 13.8 Common Errors to Avoid

- Trying to use Cartesian coordinates when drawing shapes. The screen coordinate system differs from the Cartesian coordinate system that you learned about in mathematics. In the Cartesian coordinate system, the Y coordinates decrease as you move downward. In the screen coordinate system, the Y coordinates increase as you move downward, toward the bottom of the screen.
- Specifying too many vertices for the `Polygon` class. The `Polygon` class draws a closed polygon, so it automatically draws a line segment connecting the ending vertex with the first vertex. It is a common mistake to specify the same values for both the starting vertex and the ending vertex.
- Rotating a node in the wrong direction with a `RotationTransition`. If you pass a positive angle to the `setToAngle` or the `setByAngle` methods, the rotation will be clockwise. If you pass a negative angle to either of these methods, the rotation will be counterclockwise.
- Trying to combine effects using the `Node` class's `setEffect` method. The `Node` class's `setEffect` method accepts only one effect object as an argument. If you call the `setEffect` method more than once for a particular node, the last effect object that you pass as an argument replaces any previously passed effect objects for that node. However, it is possible to combine effects using the `Effect` class's `setInput` method.
- Registering a `KeyEvent` handler with the wrong node. In most cases, you will want to register your `KeyEvent` handlers with the `Scene` object. That way, your `KeyEvent` handlers will respond to key events, regardless of which node has the input focus.

# Review Questions

## Multiple Choice and True/False

1. Line, Circle, and Rectangle are subclasses of the class.
  1. Sprite
  2. Geometry
  3. BasicShape
  4. Shape
2. Assume myCircle is a Circle object. Which of these statements causes myCircle to not be filled with a color?
  1. myCircle.setColor(null);
  2. myCircle.setStroke(null);
  3. myCircle.setFill(null);
  4. myCircle.setFill(Color.NONE);
3. The arc type causes a straight line to be drawn from one endpoint of the arc to the other endpoint.
  1. ArcType.CHORD
  2. ArcType.ROUND
  3. ArcType.OPEN

4. ArcType.CLOSED
4. The arc type causes straight lines to be drawn from each endpoint to the arc's center point. As a result, the arc will be shaped like a pie slice.

  1. ArcType.CHORD
  2. ArcType.ROUND
  3. ArcType.OPEN
  4. ArcType.CLOSED
5. The arc type causes no lines to connect the endpoints. Only the arc will be drawn.

  1. ArcType.CHORD
  2. ArcType.ROUND
  3. ArcType.OPEN
  4. ArcType.CLOSED
6. Which Node class method do you call to rotate a node about its center?

  1. rotate()
  2. rotateCenter()
  3. setRotate()
  4. centerRotate()
7. This class creates an animation in which a node moves from one position to another over a period of time.

  1. TranslateTransition

2. `MoveTransition`
  3. `MotionTransition`
  4. `PositionTransition`
8. You use this class to specify the amount of time that an animation should last.
  1. `TimeSpan`
  2. `Duration`
  3. `Clock`
  4. `AnimationTime`
9. This interpolator causes an animation to start slowly, accelerate, then slow down at the end. This is the default interpolator.
  1. `Interpolator.SLOW`
  2. `Interpolator.EASE_START_END`
  3. `Interpolator.GRADUAL`
  4. `Interpolator.EASE_BOTH`
10. You use these two classes to play an audio file.
  1. `Media` and `MediaPlayer`
  2. `Audio` and `AudioPlayer`
  3. `Sound` and `SoundPlayer`
  4. `LoadMedia` and `PlayMedia`
11. True or False: In a 640 by 480 window, the coordinates of the pixel in

the upper-left corner are (0, 0)

12. True or False: In a 640 by 480 window, the coordinates of the pixel in the lower-right corner are (640, 480)
13. True or False: If an ellipse's X-radius and Y-radius have the same value, then the ellipse will be a circle.
14. True or False: The `Node` class's `setEffect` method accepts multiple effect objects as arguments, allowing you to combine effects.
15. True or False: Any time the user releases a key, a `KEY_PRESSED` event occurs.

## Algorithm Workbench

1. Write a statement that instantiates the `Line` class, and uses the constructor to draw a line from (50, 25) to (150, 125).
2. Write a statement that instantiates the `Circle` class, and uses the constructor to draw a circle with its center point at (100, 75) and with a radius of 120.
3. Write a statement that instantiates the `Rectangle` class, and uses the constructor to draw a rectangle with its upper-left corner at (100, 200), with a width of 175, and a height of 150.
4. Write code that does the following: Instantiates the `Ellipse` class, and uses the constructor to draw an ellipse with its center located at (300, 200), an x-radius of 120 pixels, and a y-radius of 100. Then, set the ellipse's stroke color to black, and the fill color to red.
5. Write a statement that instantiates the `Polygon` class, and uses the constructor to draw a polygon with vertices at the following points: (100, 100), (200, 100), (200, 200), and (100, 200).
6. Write a statement that instantiates the `Text` class, and uses the

constructor to draw the string “Hello World”, starting at the coordinates 100, 200.

7. Assume `myBox` is a `Rectangle` object. Write the statement that rotates `myBox` 45 degrees about its center.
8. Assume `myBox` is a `Rectangle` object. Write the code that scales `myBox` to half its current size, along both the `X` and `Y` axes.
9. Write code that creates a `Circle`, with a radius of 30, and then creates a `TranslateTransition` object to animate it. The duration of the animation is 5 seconds. When the animation is played, the `circle` object will start at (0, 100), and it will move to the location (200, 150). It will take 3 seconds for the `circle` object to complete the move.
10. Write code that loads the audio file `C:\media\saxophone.wav`, and creates a `MediaPlayer` object that is ready to play the file.
11. Write code that loads the video file `meteor_shower.mp4`, and creates a `MediaViewer` object that is ready to play the file in a viewing window that is 640 by 480 in size.

## Short Answer

1. In a 640 by 480 window, what are the coordinates of the pixel in the upper-right corner of the window?
2. How would you use the `Ellipse` class to draw a circle?
3. What `TranslateTransition` class methods do you use to specify a node’s starting location?
4. What `TranslateTransition` class methods do you use to specify a node’s ending location?
5. What `RotateTransition` class method do you use to specify a node’s starting angle?

6. What `RotateTransition` class method do you use to specify a node's ending angle?
7. What `RotateTransition` class method do you use to specify the amount of rotation?
8. What `ScaleTransition` class methods do you use to specify a node's starting scale factor?
9. What `ScaleTransition` class methods do you use to specify a node's ending scale factor?
10. What two effect classes can you use to create shadow effects?
11. What three effect classes can you use to create blur effects?

# Programming Challenges

## 1. This Old House

Use the basic shapes you learned in this chapter to draw a house. Be sure to include at least two windows and a door. Feel free to draw other objects as well, such as the sky, sun, and even clouds.

## 2. Tree Age

Counting the growth rings of a tree is a good way to tell the age of a tree. Each growth ring counts as one year. Use either the `Circle` or `Ellipse` classes to draw how the growth rings of a 5-year-old tree might look. Then, using the `Text` shape class, number each growth ring starting from the center and working outward with the age in years associated with that ring.

## 3. Hollywood Star

Make your own star on the Hollywood Walk of Fame. Write a program that displays a star similar to the one shown in [Figure 13-43](#), with your name displayed in the star.

## Figure 13-43 Hollywood star



#### 4. Vehicle Outline

Using the shapes you learned about in this chapter, draw the outline of the vehicle of your choice (car, truck, airplane, and so forth).

#### 5. Solar System

Use the `Circle` class to draw each of the planets of our solar system. Draw the sun first, then each planet according to distance from the sun. (Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, and the dwarf planet, Pluto.) Label each planet using the `Text` shape class.

#### 6. Opacity Adjuster

Create an application that lets the user select an image file via a `FileChooser` dialog. The application should display the image, and provide a `Slider` control. The `Slider` control should let the user adjust the image's opacity.

#### 7. Color Adjuster

Create an application that lets the user select an image file via a `FileChooser` dialog. The application should display the image and provide four `Slider` controls. The `Slider` controls should let the user adjust the image's hue, saturation, brightness, and contrast.

## 8. Mouse Rollover

Create an application that displays an image. When the user moves the mouse pointer over the image, it should change to a second image. The second image should remain displayed as long as the mouse pointer is over it. When the user moves the mouse pointer away from the second image, it should change back to the first image.

## 9. Coin Toss

Write a program that simulates the tossing of a coin. The program should wait for the user to press a key, then generate a random number in the range of 0 through 1. If the number is 0, the program should display the image of a coin with heads facing up. If the number is 1, the program should display the image of a coin with tails facing up. (You can create your own coin images, or use the ones provided in the book's online resources, downloadable from [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).)



### VideoNote The Coin Toss Problem

## 10. Lunar Lander

The book's online resources (downloadable from [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)) provide images of a spacecraft and a background drawing of the moon's surface. Write a program that initially displays the spacecraft on the moon's surface. When the user presses the spacebar, the spacecraft should slowly lift off the surface.

## 11. Change for a Dollar Game

The book's online resources (downloadable from

[www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis)) provide images of a penny, a nickel, a dime, and a quarter. Create a game that displays each of these images, plus the text “Count Change”. The game should let the user click any of the coins, in any order. To win the game, the user must click the coins that, when added together, equal exactly one dollar. When the user clicks the “Count Change” text, the program should show the amount of money that the user clicked. If the amount equals one dollar, the program should indicate that the user won the game.

## 12. Rock, Paper, Scissors Game

Write a program that lets the user play the game of Rock, Paper, Scissors against

the computer. The program should work as follows:

1. When the program begins, a random number in the range of 0 through 2 is generated. If the number is 0, then the computer has chosen rock. If the number is 1, then the computer has chosen paper. If the number is 2, then the computer has chosen scissors. (Don’t display the computer’s choice yet.)
2. To make his or her selection, the user clicks an image on the screen. (You can find images for this game included in the book’s online resources at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).)
3. The computer’s choice is displayed.
4. A winner is selected according to the following rules:
  - If one player chooses rock and the other player chooses scissors, then rock wins. (The rock smashes the scissors.)
  - If one player chooses scissors and the other player chooses paper, then scissors wins. (Scissors cuts paper.)
  - If one player chooses paper and the other player chooses rock, then paper wins. (Paper wraps rock.)

- If both players make the same choice, the game is a tie.

# **Chapter 14 Recursion**

## **Topics**

1. [14.1 Introduction to Recursion](#)
2. [14.2 Solving Problems with Recursion](#)
3. [14.3 Examples of Recursive Methods](#)
4. [14.4 A Recursive Binary Search Method](#)
5. [14.5 The Towers of Hanoi](#)
6. [14.6 Common Errors to Avoid](#)

# 14.1 Introduction to Recursion

## Concept:

A recursive method is a method that calls itself.

You have seen instances of methods calling other methods. Method A can call method B, which can then call method C. It's also possible for a method to call itself. A method that calls itself is a *recursive method*. Look at the message method in [Code Listing 14-1](#).

## Code Listing 14-1 (EndlessRecursion.java)

```
1 /**
2 * This class has a recursive method.
3 */
4
5 public class EndlessRecursion
6 {
7 public static void message()
8 {
9 System.out.println("This is a recursive method.");
10 message();
11 }
12 }
```

This method displays the string “This is a recursive method.”, then calls itself. Each time it calls itself, the cycle is repeated. Can you see a problem with the method? There’s no way to stop the recursive calls. This method is like an infinite loop, because there is no code to stop it from repeating.

Like a loop, a recursive method must have some way to control the number of times it repeats. The class in [Code Listing 14-2](#) has a modified version of the message method. It passes an integer argument, which holds the number of times the method should call itself.

## Code Listing 14-2 (Recursive.java)

```
1 /**
2 * This class has a recursive method, message, that displays
3 * a message n times.
4 */
5
6 public class Recursive
7 {
8 public static void message(int n)
9 {
10 if (n > 0)
11 {
12 System.out.println("This is a recursive method.");
13 message(n - 1);
14 }
15 }
16 }
```

This method contains an `if` statement that controls the repetition. As long as the `n` parameter is greater than zero, the method displays the message and calls itself again. Each time it calls itself, it passes `n - 1` as the argument. For example, look at the program in [Code Listing 14-3](#).

## Code Listing 14-3 (RecursionDemo.java)

```
1 /**
2 * This class demonstrates the Recursive.message method.
3 */
4
5 public class RecursionDemo
6 {
7 public static void main(String[] args)
```

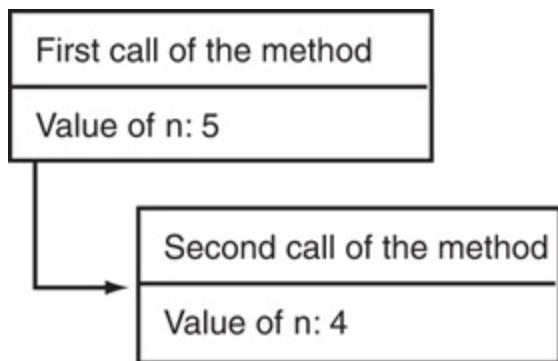
```
8 {
9 Recursive.message(5);
10 }
11 }
```

## Program Output

This is a recursive method.  
This is a recursive method.

In line 9, the `main` method in this class calls the `Recursive.message` method with argument 5, which causes the method to call itself five times. The first time the method is called, the `if` statement displays the message then calls itself with 4 as the argument. [Figure 14-1](#) illustrates this.

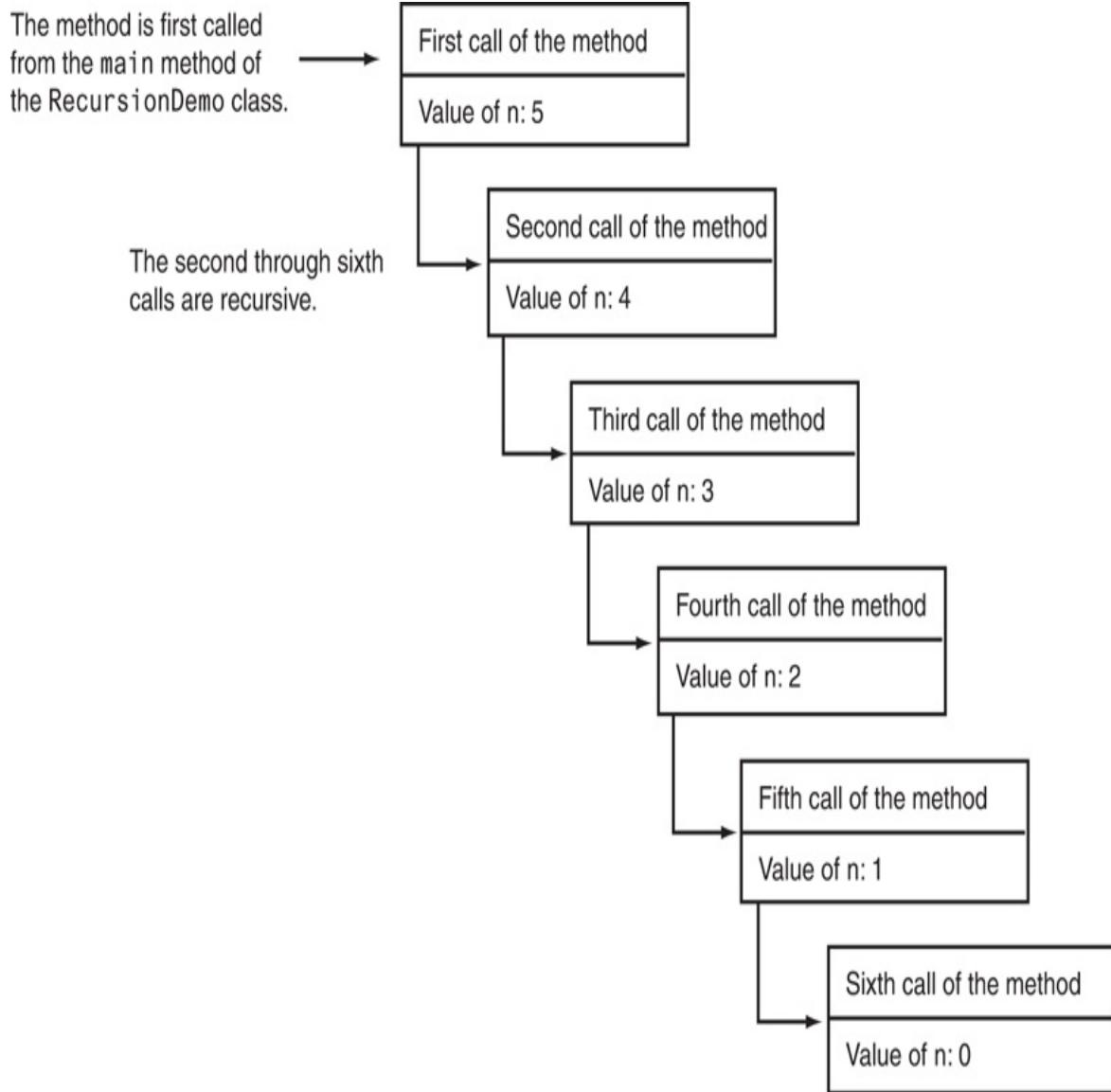
## Figure 14-1 First two calls of the method



The diagram in [Figure 14-1](#) illustrates two separate calls of the `message` method. Each time the method is called, a new instance of the `n` parameter is created in memory. The first time the method is called, the `n` parameter is set to 5. When the method calls itself, a new instance of `n` is created, and the value 4 is passed into it. This cycle repeats until, finally, zero is passed to the method. This is illustrated in [Figure 14-2](#).

As you can see from [Figure 14-2](#), the method is called a total of six times. The first time it is called from the `main` method of the `RecursionDemo` class, and the other five times it calls itself. The number of times that a method calls itself is known as the *depth of recursion*. In this example, the depth of recursion is five. When the method reaches its sixth call, the `n` parameter is set to 0. At that point, the `if` statement's conditional expression is false, so the method returns. Control of the program returns from the sixth instance of the method to the point in the fifth instance directly after the recursive method call. This is illustrated in [Figure 14-3](#).

## Figure 14-2 Total of six calls to the message method



[Figure 14-2 Full Alternative Text](#)

**Figure 14-3 Control returns to the point after the recursive method call**

```
public static void message(int n)
{
 if (n > 0)
 {
 System.out.println("This is a recursive method.");
 message(n - 1);
 }
}
```

Recursive method call → Control returns here from the recursive call.  
There are no more statements to execute in this method, so the method returns.

[Figure 14-3 Full Alternative Text](#)

Because there are no more statements to be executed after the method call, the fifth instance of the method returns control of the program back to the fourth instance. This repeats until all instances of the method return.

# 14.2 Solving Problems with Recursion

## Concept:

A problem can be solved with recursion if it can be broken down into successive smaller problems that are identical to the overall problem.

The Recursive and RecursionDemo classes shown in the previous section demonstrate the mechanics of a recursive method. Recursion can be a powerful tool for solving repetitive problems, and is an important topic in upper-level computer science courses. What might not be clear to you yet is how to use recursion to solve a problem.



### VideoNote Reducing a Problem with Recursion

First, it should be noted that recursion is never absolutely required to solve a problem. Any problem that can be solved recursively can also be solved iteratively, with a loop. In fact, recursive algorithms are usually less efficient than iterative algorithms. This is because a method call requires several actions to be performed by the JVM. These actions include allocating memory for parameters and local variables, and storing the address of the program location where control returns after the method terminates. These actions, which are sometimes referred to as *overhead*, take place with each method call. Such overhead is not necessary with a loop.

Some repetitive problems, however, are more easily solved with recursion than with iteration. Whereas an iterative algorithm might result in faster execution time, the programmer might be able to design a recursive algorithm

faster.

In general, a recursive method works like this:

- If the problem can be solved now, without recursion, then the method solves it and returns.
- If the problem cannot be solved now, then the method reduces it to a smaller but similar problem and calls itself to solve the smaller problem.

In order to apply this approach, we first identify at least one case in which the problem can be solved without recursion. This is known as the *base case*. Second, we determine a way to solve the problem in all other circumstances using recursion. This is called the *recursive case*. In the recursive case, we must always reduce the problem to a smaller version of the original problem. By reducing the problem with each recursive call, the base case will eventually be reached, and the recursion will stop.

Let's take an example from mathematics to examine an application of recursion. In mathematics, the notation  $n!$  represents the factorial of the number  $n$ . The factorial of a nonnegative number can be defined by the following rules:

$$\begin{array}{ll} \text{If } n = 0 \text{ then} & n! = 1 \\ \text{If } n > 0 \text{ then} & n! = 1 \times 2 \times 3 \times \dots \times n \end{array}$$

Let's replace the notation  $n!$  with  $\text{factorial}(n)$ , which looks a bit more like computer code, and rewrite these rules as:

$$\begin{array}{ll} \text{If } n = 0 \text{ then} & \text{factorial}(n) = 1 \\ \text{If } n > 0 \text{ then} & \text{factorial}(n) = 1 \times 2 \times 3 \times \dots \times n \end{array}$$

These rules state that when  $n$  is 0, its factorial is 1. When  $n$  is greater than 0, its factorial is the product of all the positive integers from 1 up to  $n$ . For instance,  $\text{factorial}(6)$  is calculated as  $1 \times 2 \times 3 \times 4 \times 5 \times 6$ .

When designing a recursive algorithm to calculate the factorial of any number, we first identify the base case, which is the part of the calculation that we can solve without recursion. That is the case where  $n$  is equal to 0:

```
If n = 0 then factorial(n) = 1
```

This tells how to solve the problem when  $n$  is equal to 0, but what do we do when  $n$  is greater than 0? That is the recursive case, or the part of the problem that we use recursion to solve. This is how we express it:

```
If n > 0 then factorial(n) = n × factorial(n - 1)
```

This states that if  $n$  is greater than 0, the factorial of  $n$  is  $n$  times the factorial of  $n - 1$ . Notice how the recursive call works on a reduced version of the problem,  $n - 1$ . So, our recursive rule for calculating the factorial of a number might look like this:

```
If n = 0 then factorial(n) = 1
If n > 0 then factorial(n) = n × factorial(n - 1)
```

The following code shows how this might be implemented in a Java method:

```
private static int factorial(int n)
{
 if (n == 0)
 return 1; // Base case
 else
 return n * factorial(n - 1);
}
```

The program in [Code Listing 14-4](#) demonstrates the method.

## Code Listing 14-4 (FactorialDemo.java)

```
1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the recursive factorial method.
5 */
6
7 public class FactorialDemo
8 {
9 public static void main(String[] args)
```

```

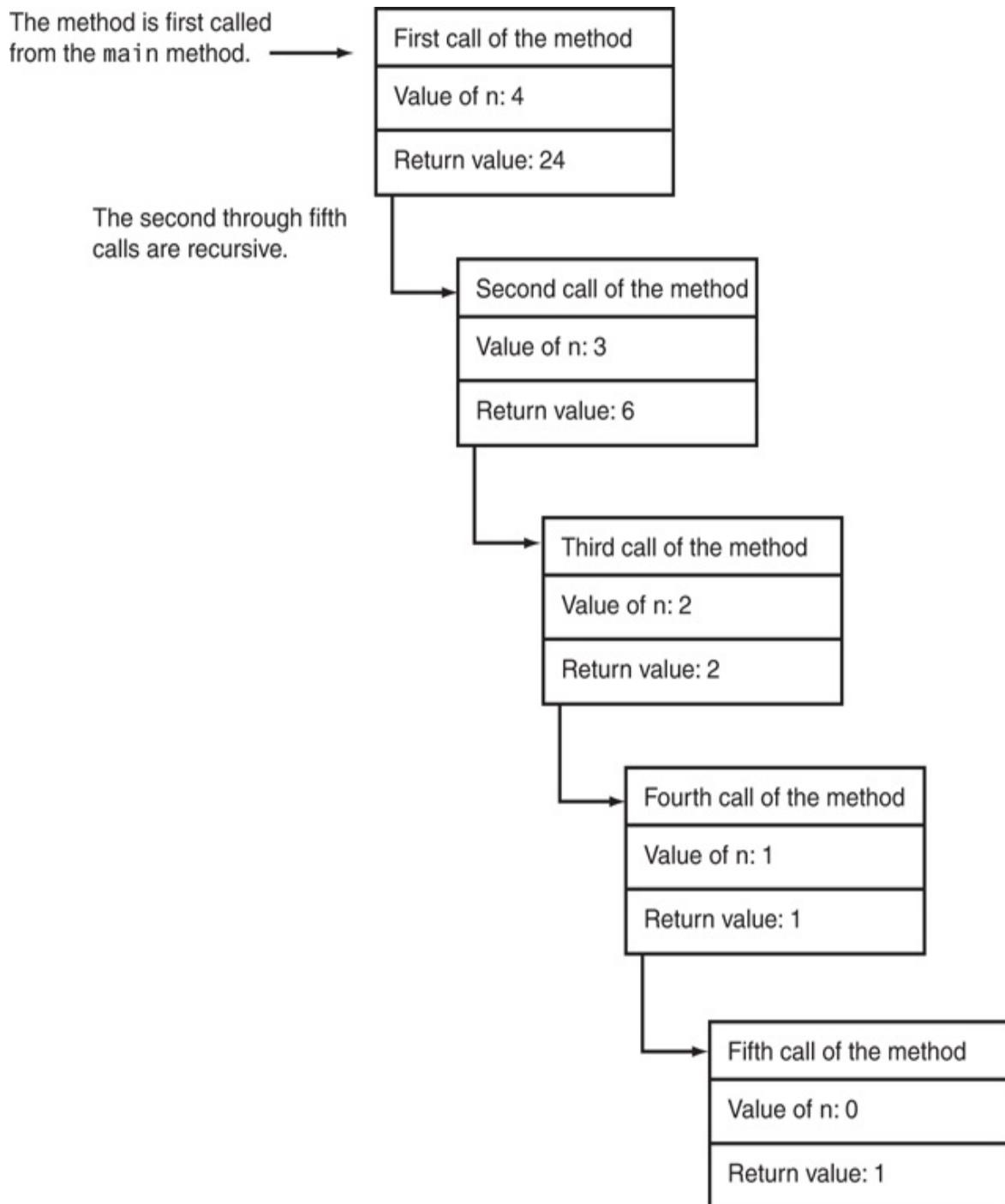
10 {
11 int number; // To hold a number
12
13 // Create a Scanner object for keyboard input.
14 Scanner keyboard = new Scanner(System.in);
15
16 // Get a number from the user.
17 System.out.print("Enter a nonnegative integer: ");
18 number = keyboard.nextInt();
19
20 // Display the factorial.
21 System.out.println(number + "! is " + factorial(number));
22 }
23
24 /**
25 * Recursive factorial method. This method returns the
26 * factorial of its argument, which is assumed to be a
27 * nonnegative number.
28 */
29
30 private static int factorial(int n)
31 {
32 if (n == 0)
33 return 1; // Base case
34 else
35 return n * factorial(n - 1);
36 }
37 }
```

### **Program Output with Example Input Shown in Bold**

Enter a nonnegative integer: **4**   
4! is 24

In the example run of the program, the factorial method is called with the argument 4 passed into `n`. Because `n` is not equal to 0, the `if` statement's `else` clause executes the statement in line 35. Although this is a `return` statement, it does not immediately return. Before the return value can be determined, the value of `factorial(n - 1)` must be determined. The `factorial` method is called recursively until the fifth call, in which the `n` parameter will be set to zero. The diagram in [Figure 14-4](#) illustrates the value of `n` and the return value during each call of the method.

# Figure 14-4 Recursive calls to the factorial method



### Figure 14-4 Full Alternative Text

This diagram illustrates why a recursive algorithm must reduce the problem with each recursive call. Eventually, the recursion has to stop in order for a solution to be reached. If each recursive call works on a smaller version of the problem, then the recursive calls work toward the base case. The base case does not require recursion, so it stops the chain of recursive calls.

Usually, a problem is reduced by making the value of one or more parameters smaller with each recursive call. In our `factorial` method, the value of the parameter `n` gets closer to 0 with each recursive call. When the parameter reaches 0, the method returns a value without making another recursive call.

## Direct and Indirect Recursion

The examples we have discussed so far show recursive methods that directly call themselves. This is known as *direct recursion*. There is also the possibility of creating *indirect recursion* in a program. This occurs when method A calls method B, which in turn calls method A. There can even be several methods involved in the recursion. For example, method A could call method B, which could call method C, which calls method A.



### Checkpoint

1. 14.1 It is said that a recursive algorithm has more overhead than an iterative algorithm. What does this mean?
2. 14.2 What is a base case?
3. 14.3 What is a recursive case?
4. 14.4 What causes a recursive algorithm to stop calling itself?
5. 14.5 What is direct recursion? What is indirect recursion?

## 14.3 Examples of Recursive Methods

### Summing a Range of Array Elements with Recursion

In this example, we look at a method, `rangeSum`, that uses recursion to sum a range of array elements. The method takes the following arguments: an `int` array containing the range of elements to be summed, an `int` specifying the starting element of the range, and an `int` specifying the ending element of the range. Here is an example of how the method might be used:

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int sum;
sum = rangeSum(numbers, 3, 7);
```

This code specifies that `rangeSum` should return the sum of elements three through seven in the `numbers` array. The return value, which in this case would be 30, is stored in `sum`. Here is the definition of the `rangeSum` method:

```
public static int rangeSum(int[] array, int start, int end)
{
 if (start > end)
 return 0;
 else
 return array[start] + rangeSum(array, start + 1, end);
}
```

This method's base case is when the `start` parameter is greater than the `end` parameter. If this is true, the method returns the value 0. Otherwise, the method executes the following statement:

```
return array[start] + rangeSum(array, start + 1, end);
```

This statement returns the sum of array[start] plus the return value of a recursive call. Notice in the recursive call, the starting element in the range is start + 1. In essence, this statement says “return the value of the first element in the range plus the sum of the rest of the elements in the range.” The program in [Code Listing 14-5](#) demonstrates the method.

## Code Listing 14-5 (RangeSum.java)

```
1 /**
2 * This program demonstrates the recursive rangeSum method.
3 */
4
5 public class RangeSum
6 {
7 /**
8 * main method
9 */
10
11 public static void main(String[] args)
12 {
13 int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
14
15 System.out.print("The sum of elements 2 through 5 is " +
16 rangeSum(numbers, 2, 5));
17 }
18
19 /**
20 * The rangeSum method returns the sum of a specified
21 * range of elements in array. The start parameter
22 * specifies the starting element and the end parameter
23 * specifies the ending parameter.
24 */
25
26 public static int rangeSum(int[] array, int start, int end)
27 {
28 if (start > end)
29 return 0;
30 else
31 return array[start] + rangeSum(array, start + 1, end);
32 }
33 }
```

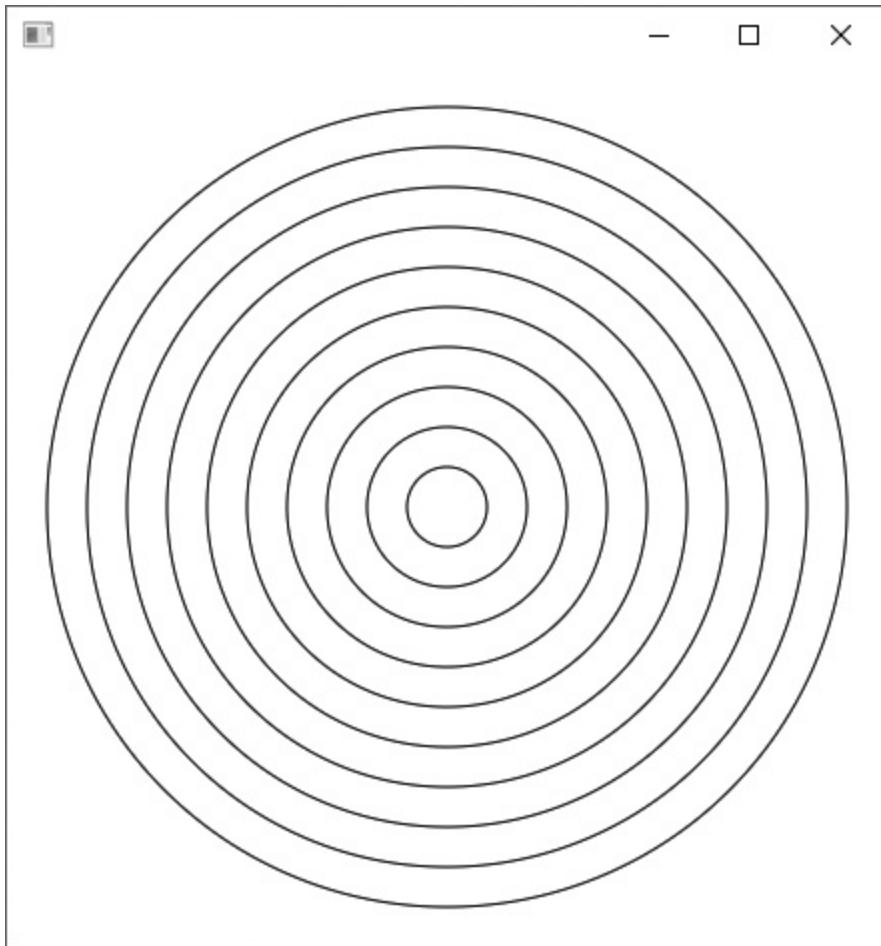
### Program Output

The sum of elements 2 through 5 is 18

## Drawing Concentric Circles

In this example, we look at the `Circles` application, which uses recursion to draw concentric circles. Concentric circles are circles that share a common center point. [Figure 14-5](#) shows the application's output. The code is shown in [Code Listing 14-6](#).

### Figure 14-5 Circles application



### Code Listing 14-6 (`Circles.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.Pane;
5 import javafx.scene.shape.Circle;
6 import javafx.scene.paint.Color;
7
8 public class Circles extends Application
9 {
10 public static void main(String[] args)
11 {
12 launch(args);
13 }
14
15 @Override
16 public void start(Stage primaryStage)
17 {
18 // Constants for the scene size
19 final double SCENE_WIDTH = 440.0;
20 final double SCENE_HEIGHT = 440.0;
21
22 // Constants for the starting values
23 final double CENTER_X = 220.0;
24 final double CENTER_Y = 220.0;
25 final double RAD = 20.0;
26 final int NUM_CIRCLES = 10;
27
28 // Create an empty Pane.
29 Pane pane = new Pane();
30
31 // Recursively add 10 circles to the Pane.
32 drawCircles(pane, NUM_CIRCLES, CENTER_X, CENTER_Y, RAD);
33
34 // Create a Scene and display it.
35 Scene scene = new Scene(pane, SCENE_WIDTH, SCENE_HEIGHT);
36 primaryStage.setScene(scene);
37 primaryStage.show();
38 }
39
40 /**
41 * The drawCircles method draws concentric circles.
42 * It accepts the following arguments:
43 * * p -- a Pane object to add the circles to
44 * * n -- the number of circles to draw
45 * * x -- x coordinate of the circle's center point
46 * * y -- y coordinate of the circle's center point
47 * * rad -- the circle's radius
48 */

```

```

49
50 private void drawCircles(Pane p, int n, double x, double y, d
51 {
52 if (n > 0)
53 {
54 Circle circle = new Circle(x, y, rad); // Create the circ
55 circle.setStroke(Color.BLACK); // Line color is b
56 circle.setFill(null); // No fill color
57 p.getChildren().addAll(circle); // Add the it to t
58 drawCircles(p, n - 1, x, y, rad + 20); // Draw the next c
59 }
60 }
61 }
```

The `drawCircles` method, which is called from the application's start method, uses recursion to draw the concentric circles. The `n` parameter holds the number of circles to draw. If this parameter is set to 0, the method has reached its base case. Otherwise, line 54 creates a `Circle` object at the specified center point, using the specified radius. Line 55 sets the `Circle`'s stroke color to black, and line 56 sets the `Circle`'s fill color to `null` (the circle will not be filled with a color). Line 57 adds the `Circle` object to the `Pane`. Line 58 makes a recursive call to the `drawCircles` method with the parameters adjusted for the next circle.

## The Fibonacci Series

Some mathematical problems are designed to be solved recursively. One well-known example is the calculation of *Fibonacci numbers*. The Fibonacci numbers, named after the Italian mathematician Leonardo Fibonacci (born circa 1170), are the following sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

Notice after the second number, each number in the series is the sum of the two previous numbers. The Fibonacci series can be defined as:

|                    |                                                         |
|--------------------|---------------------------------------------------------|
| If $n = 0$ then    | $\text{Fib}(n) = 0$                                     |
| If $n = 1$ then    | $\text{Fib}(n) = 1$                                     |
| If $n \geq 2$ then | $\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$ |

A recursive Java method to calculate the  $n$ th number in the Fibonacci series is shown here:

```
public static int fib(int n)
{
 if (n == 0)
 return 0;
 else if (n == 1)
 return 1;
 else
 return fib(n - 1) + fib(n - 2);
}
```

Notice this method actually has two base cases: when  $n$  is less than 0, and when  $n$  is equal to 1. In either case, the method returns a value without making a recursive call. The program in [Code Listing 14-7](#) demonstrates this method by displaying the first 10 numbers in the Fibonacci series.

## Code Listing 14-7 (FibNumbers.java)

```
1 /**
2 * This program demonstrates the recursive fib method.
3 */
4
5 public class FibNumbers
6 {
7 /**
8 * main method
9 */
10
11 public static void main(String[] args)
12 {
13 System.out.println("The first 10 numbers in the " +
14 "Fibonacci series are:");
15
16 for (int i = 0; i < 10; i++)
17 System.out.print(fib(i) + " ");
18
19 System.out.println();
20 }
21
```

```

22 /**
23 * The fib method returns the nth
24 * Fibonacci number.
25 */
26
27 public static int fib(int n)
28 {
29 if (n == 0)
30 return 0;
31 else if (n == 1)
32 return 1;
33 else
34 return fib(n - 1) + fib(n - 2);
35 }
36 }
```

### **Program Output**

The first 10 numbers in the Fibonacci series are:  
 0 1 1 2 3 5 8 13 21 34

## **Finding the Greatest Common Divisor**

Our next example of recursion is the calculation of the greatest common divisor, or GCD, of two numbers. The GCD of two positive integers,  $x$  and  $y$ , is:

if  $y$  divides  $x$  evenly, then  $\text{gcd}(x, y) = y$   
 Otherwise,  $\text{gcd}(x, y) = \text{gcd}(y, \text{remainder of } x/y)$

This definition states that the GCD of  $x$  and  $y$  is  $y$  if  $x/y$  has no remainder. This is the base case. Otherwise, the answer is the GCD of  $y$  and the remainder of  $x/y$ . The program in [Code Listing 14-8](#) shows a recursive method for calculating the GCD.

## **Code Listing 14-8 (GCDdemo.java)**

```

1 import java.util.Scanner;
2
3 /**
4 * This program demonstrates the recursive gcd method.
5 */
6
7 public class GCDdemo
8 {
9 /**
10 * main method
11 */
12
13 public static void main(String[] args)
14 {
15 int num1, num2; // Two numbers
16
17 // Create a Scanner object for keyboard input.
18 Scanner keyboard = new Scanner(System.in);
19
20 // Get two numbers from the user.
21 System.out.print("Enter an integer: ");
22 num1 = keyboard.nextInt();
23 System.out.print("Enter another integer: ");
24 num2 = keyboard.nextInt();
25
26 // Display the GCD.
27 System.out.println("The greatest common divisor " +
28 "of these two numbers is " +
29 gcd(num1, num2));
30 }
31
32 /**
33 * The gcd method returns the greatest common divisor
34 * of the arguments passed into x and y.
35 */
36
37 public static int gcd(int x, int y)
38 {
39 if (x % y == 0)
40 return y;
41 else
42 return gcd(y, x % y);
43 }
44 }
```

## Program Output with Example Input Shown in Bold

Enter an integer: **49**

Enter another integer: **28**

The greatest common divisor of these two numbers is 7

# 14.4 A Recursive Binary Search Method

## Concept:

The recursive binary search algorithm is more elegant and easier to understand than its iterative version.

In [Chapter 7](#), you learned about the binary search algorithm and saw an iterative example written in Java. The binary search algorithm can also be implemented recursively. For example, the procedure can be expressed as:

- *If array[middle] equals the search value, then the value is found.*
- *Else if array[middle] is less than the search value, perform a binary search on the upper half of the array.*
- *Else if array[middle] is greater than the search value, perform a binary search on the lower half of the array.*

When you compare the recursive algorithm to its iterative counterpart, it becomes evident that the recursive version is much more elegant and easier to understand. The recursive binary search algorithm is also a good example of repeatedly breaking a problem down into smaller pieces until it is solved. Here is the code for the method:

```
public static int binarySearch(int[] array, int first, int last,
{
 int middle; // Mid point of search

 // Test for the base case where the value is not found.
 if (first > last)
 return -1;
```

```

// Calculate the middle position.
middle = (first + last) / 2;
// Search for the value.
if (array[middle] == value)
 return middle;
else if (array[middle] < value)
 return binarySearch(array, middle + 1, last, value);
else
 return binarySearch(array, first, middle - 1, value);
}

```

The first parameter, `array`, is the array to be searched. The next parameter, `first`, holds the subscript of the first element in the search range (the portion of the array to be searched). The next parameter, `last`, holds the subscript of the last element in the search range. The last parameter, `value`, holds the value to be searched for. Like the iterative version, this method returns the subscript of the value if it is found, or `-1` if the value is not found. [Code](#) [Listing 14-9](#) demonstrates the method.

## Code Listing 14-9 (RecursiveBinarySearch.java)

```

1 import java.util.Scanner;
2
3 /**
4 * This demonstrates the recursive binary search method.
5 */
6
7 public class RecursiveBinarySearch
8 {
9 public static void main(String [] args)
10 {
11 // The values in the following array are sorted
12 // in ascending order.
13 int numbers[] = {101, 142, 147, 189, 199, 207, 222,
14 234, 289, 296, 310, 319, 388, 394,
15 417, 429, 447, 521, 536, 600};
16
17 int result; // Result of the search
18 int searchValue; // Value to search for
19 String again; // User input

```

```
20
21 // Create a Scanner object for keyboard input.
22 Scanner keyboard = new Scanner(System.in);
23
24 do
25 {
26 // Get a value to search for.
27 System.out.print("Enter a value to search for: ");
28 searchValue = keyboard.nextInt();
29
30 // Search for the value
31 result = binarySearch(numbers, 0,
32 (numbers.length - 1), searchValue);
33
34 // Display the results.
35 if (result == -1)
36 {
37 System.out.println(searchValue +
38 " was not found.");
39 }
40 else
41 {
42 System.out.println(searchValue +
43 " was found at " +
44 "element " + result);
45 }
46
47 // Consume the remaining newline.
48 keyboard.nextLine();
49
50 // Does the user want to search again?
51 System.out.print("Do you want to search again? " +
52 "(Y or N): ");
53 again = keyboard.nextLine();
54
55 } while (again.charAt(0) == 'y' || again.charAt(0) == 'Y');
56
57 /**
58 * The binarySearch method performs a binary search on an
59 * integer array. The array is searched for the number passed
60 * to value. If the number is found, its array subscript is
61 * returned. Otherwise, -1 is returned indicating the value w
62 * not found in the array.
63 */
64
65
66 public static int binarySearch(int[] array, int first,
67 int last, int value)
```

```
68 {
69 int middle; // Mid-point of search
70
71 // Test for the base case where the value is not found.
72 if (first > last)
73 return -1;
74
75 // Calculate the middle position.
76 middle = (first + last) / 2;
77
78 // Search for the value.
79 if (array[middle] == value)
80 return middle;
81 else if (array[middle] < value)
82 return binarySearch(array, middle + 1, last, value);
83 else
84 return binarySearch(array, first, middle - 1, value);
85 }
86 }
```

## Program Output with Example Input Shown in Bold

Enter a value to search for: **289**

289 was found at element 8

Do you want to search again? (Y or N): **y**

Enter a value to search for: **388**

388 was found at element 12

Do you want to search again? (Y or N): **y**

Enter a value to search for: **101**

101 was found at element 0

Do you want to search again? (Y or N): **y**

Enter a value to search for: **999**

999 was not found.

Do you want to search again? (Y or N): **n**

See Appendix J, available at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis), for a discussion of the recursive QuickSort algorithm.

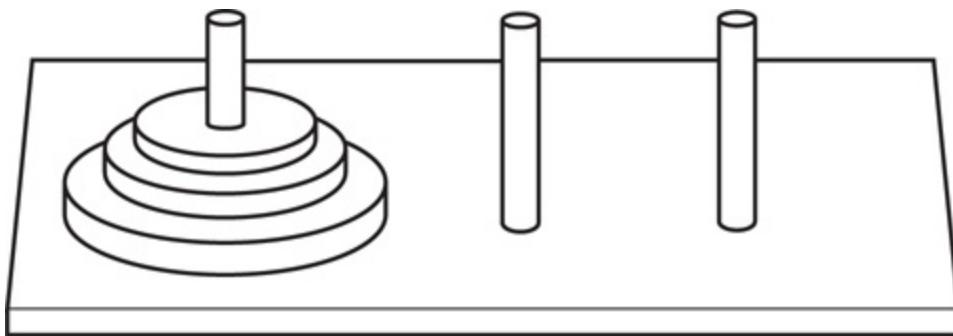
## 14.5 The Towers of Hanoi

### Concept:

The repetitive steps involved in solving the Towers of Hanoi game can be easily implemented in a recursive algorithm.

The Towers of Hanoi is a mathematical game often used to illustrate the power of recursion. The game uses three pegs and a set of discs with holes through their centers. The discs are stacked on one of the pegs as shown in [Figure 14-6](#).

**Figure 14-6 The pegs and discs in the Towers of Hanoi game**



Notice the discs are stacked on the leftmost peg, in order of size with the largest disc at the bottom. The game is based on a legend where a group of monks in a temple in Hanoi have a similar set of pegs with 64 discs. The task for the monks is to move the discs from the first peg to the third peg. The middle peg can be used as a temporary holder. Furthermore, the monks must follow these rules while moving the discs:

- Only one disc can be moved at a time.
- A disc cannot be placed on top of a smaller disc.
- All discs must be stored on a peg except while being moved.

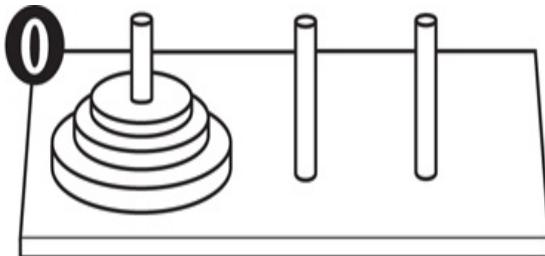
According to the legend, when the monks have moved all of the discs from the first peg to the last peg, the world will come to an end.

To play the game, you must move all of the discs from the first peg to the third peg, following the same rules as the monks. Let's look at some example solutions to this game, for different numbers of discs. If you have only one disc, the solution to the game is simple: move the disc from peg 1 to peg 3. If you have two discs, the solution requires three moves:

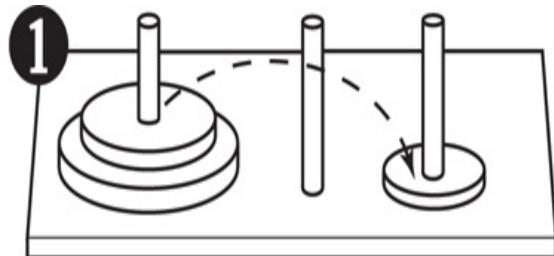
- Move disc 1 to peg 2.
- Move disc 2 to peg 3.
- Move disc 1 to peg 3.

Notice this approach uses peg 2 as a temporary location. The complexity of the moves continues to increase as the number of discs increase. To move three discs requires the seven moves shown in [Figure 14-7](#).

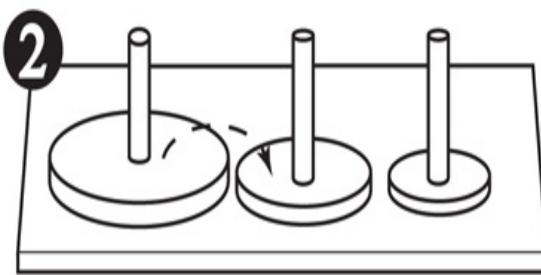
## Figure 14-7 Steps for moving three pegs



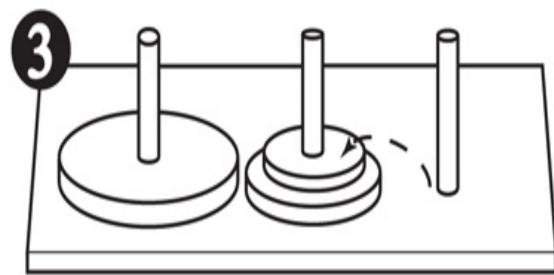
Original setup.



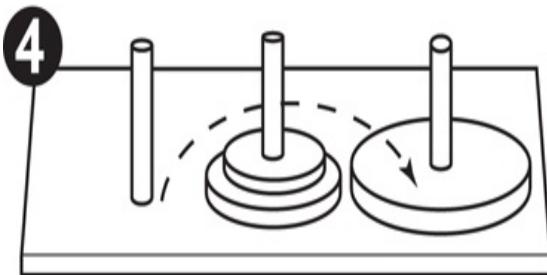
First move: Move disc 1 to peg 3.



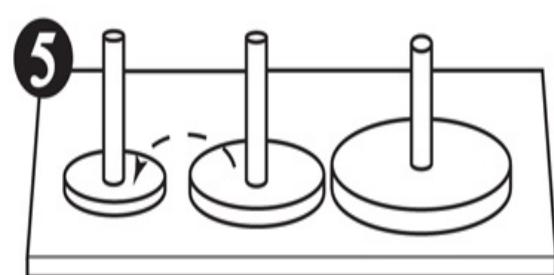
Second move: Move disc 2 to peg 2.



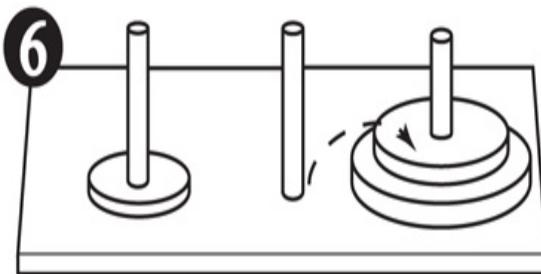
Third move: Move disc 1 to peg 2.



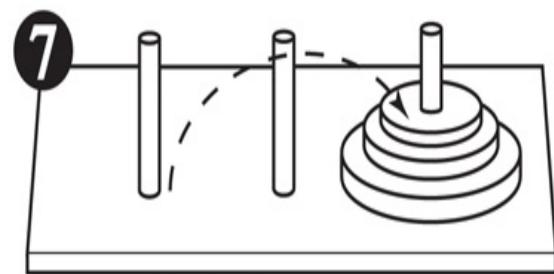
Fourth move: Move disc 3 to peg 3.



Fifth move: Move disc 1 to peg 1.



Sixth move: Move disc 2 to peg 3.



Seventh move: Move disc 1 to peg 3.

[Figure 14-7 Full Alternative Text](#)

The following statement describes the overall solution to the problem:

*Move  $n$  discs from peg 1 to peg 3 using peg 2 as a temporary peg.*

The following algorithm can be used as the basis of a recursive method that simulates the solution to the game. Notice in this algorithm we use the

variables *A*, *B*, and *C* to hold peg numbers.

*To move n discs from peg A to peg C, using peg B as a temporary peg*  
*If n > 0 then*  
    *Move n - 1 discs from peg A to peg B, using peg C as a temporary peg*  
    *Move the remaining disc from peg A to peg C.*  
    *Move n - 1 discs from peg B to peg C, using peg A as a temporary peg*  
*End If*

The base case for the algorithm is reached when there are no more discs to move. The following code is for a method that implements this algorithm. Note the method does not actually move anything, but displays instructions indicating all of the disc moves to make.

```
private void moveDiscs(int num, int fromPeg, int toPeg, int tempPeg)
{
 if (num > 0)
 {
 moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
 System.out.println("Move a disc from peg " + fromPeg +
 " to peg " + toPeg);
 moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
 }
}
```

This method accepts arguments into the following four parameters:

- num**     The number of discs to move.
- fromPeg** The peg to move the discs from.
- toPeg**    The peg to move the discs to.
- tempPeg** The peg to use as a temporary peg.

If **num** is greater than 0, then there are discs to move. The first recursive call is:

```
moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
```

This statement is an instruction to move all but one disc from **fromPeg** to **tempPeg**, using **toPeg** as a temporary peg. The next statement is:

```
System.out.println("Move a disc from peg " + fromPeg +
 " to peg " + toPeg);
```

This simply displays a message indicating that a disc should be moved from fromPeg to toPeg. Next, another recursive call is executed:

```
moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
```

This statement is an instruction to move all but one disc from tempPeg to toPeg, using fromPeg as a temporary peg. [Code Listing 14-10](#) shows the Hanoi class, which uses this method.

## Code Listing 14-10 (Hanoi.java)

```
1 /**
2 * This class displays a solution to the Towers of
3 * Hanoi game.
4 */
5
6 public class Hanoi
7 {
8 private int numDiscs; // Number of discs
9
10 /**
11 * Constructor. The argument is the number of
12 * discs to use.
13 */
14
15 public Hanoi(int n)
16 {
17 // Assign the number of discs.
18 numDiscs = n;
19
20 // Move the number of discs from peg 1 to peg 3
21 // using peg 2 as a temporary storage location.
22 moveDiscs(numDiscs, 1, 3, 2);
23 }
24
25 /**
26 * The moveDiscs method accepts the number of
27 * discs to move, the peg to move from, the peg
28 * to move to, and the temporary peg as arguments.
29 * It uses recursion to display the necessary
30 * disc moves.
31 */
32}
```

```

33 private void moveDiscs(int num, int fromPeg,
34 int toPeg, int tempPeg)
35 {
36 if (num > 0)
37 {
38 moveDiscs(num - 1, fromPeg, tempPeg, toPeg);
39 System.out.println("Move a disc from peg " +
40 fromPeg + " to peg " + toPeg);
41 moveDiscs(num - 1, tempPeg, toPeg, fromPeg);
42 }
43 }
44 }
```

The class constructor accepts an argument that is the number of discs to use in the game. It assigns this value to the numDiscs field, then calls the moveDiscs method in line 22. In a nutshell, this statement is an instruction to move all the discs from peg 1 to peg 3, using peg 2 as a temporary peg. The program in [Code Listing 14-11](#) demonstrates the class. It displays the instructions for moving three discs.

## Code Listing 14-11 (HanoiDemo.java)

```

1 /**
2 * This class demonstrates the Hanoi class, which
3 * displays the steps necessary to solve the Towers
4 * of Hanoi game.
5 */
6
7 public class HanoiDemo
8 {
9 static public void main(String[] args)
10 {
11 Hanoi towersOfHanoi = new Hanoi(3);
12 }
13 }
```

### Program Output

```

Move a disc from peg 1 to peg 3
Move a disc from peg 1 to peg 2
Move a disc from peg 3 to peg 2
```

Move a disc from peg 1 to peg 3  
Move a disc from peg 2 to peg 1  
Move a disc from peg 2 to peg 3  
Move a disc from peg 1 to peg 3

# 14.6 Common Errors to Avoid

The following list describes several errors that are commonly made when learning this chapter's topics:

- Not coding a base case. When the base case is reached, a recursive method stops calling itself. Without a base case, the method will continue to infinitely call itself.
- Not reducing the problem with each recursive call. Unless the problem is reduced (which usually means that the value of one or more critical parameters is reduced) with each recursive call, the method will not reach the base case. If the base case is not reached, the method will call itself infinitely.
- Writing the recursive call in such a way that the base case is never reached. You might have a base case and a recursive case that reduces the problem, but if the calculations are not performed in such a way that the base case is ultimately reached, the method will call itself infinitely.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. A method is called once from a program's `main` method, and then it calls itself four times. The depth of recursion is .
  1. one
  2. four
  3. five
  4. nine
2. This is the part of a problem that can be solved without recursion.
  1. base case
  2. solvable case
  3. known case
  4. iterative case
3. This is the part of a problem that is solved with recursion.
  1. base case
  2. iterative case
  3. unknown case
  4. recursion case

4. This is when a method explicitly calls itself.
  1. explicit recursion
  2. modal recursion
  3. direct recursion
  4. indirect recursion
5. This is when method A calls method B, which calls method A.
  1. implicit recursion
  2. modal recursion
  3. direct recursion
  4. indirect recursion
6. This refers to the actions taken internally by the JVM when a method is called.
  1. overhead
  2. set up
  3. clean up
  4. synchronization
7. True or False: An iterative algorithm will usually run faster than an equivalent recursive algorithm.
8. True or False: Some problems can be solved only through recursion.
9. True or False: It is not necessary to have a base case in all recursive algorithms.

10. True or False: In the base case, a recursive method calls itself with a smaller version of the original problem.

## Find the Error

1. Find the error in the following program:

```
public class FindTheError
{
 public static void main(String[] args)
 {
 myMethod(0);
 }

 public static void myMethod(int num)
 {
 System.out.print(num + " ");
 myMethod(num + 1);
 }
}
```

## Algorithm Workbench

1. Write a method that accepts a `String` as an argument. The method should use recursion to display each individual character in the `String`.
2. Modify the method you wrote in Question 1 so it displays the `String` backwards.
3. What will the following program display?

```
public class Checkpoint
{
 public static void main(String[] args)
 {
 int num = 0;
 showMe(num);
 }

 public static void showMe(int arg)
```

```

{
 if (arg < 10)
 showMe(arg + 1);
 else
 System.out.println(arg);
}
}

```

4. What will the following program display?

```

public class ReviewQuestion4
{
 public static void main(String[] args)
 {
 int num = 0;
 showMe(num);
 }

 public static void showMe(int arg)
 {
 System.out.println(arg);
 if (arg < 10)
 showMe(arg + 1);
 }
}

```

5. What will the following program display?

```

public class ReviewQuestion5
{
 public static void main(String[] args)
 {
 int x = 10;
 System.out.println(myMethod(x));
 }

 public static int myMethod(int num)
 {
 if (num <= 0)
 return 0;
 else
 return myMethod(num - 1) + num;
 }
}

```

6. Convert the following iterative method to one that uses recursion:

```
public static void sign(int n)
{
 while (n > 0)
 {
 System.out.println("No Parking");
 n--;
 }
}
```

7. Write an iterative version (using a loop instead of recursion) of the factorial method shown in this chapter.

## Short Answer

1. What is the difference between an iterative algorithm and a recursive algorithm?
2. What is a recursive algorithm's base case? What is the recursive case?
3. What is the base case of each of the recursive methods listed in Algorithm Workbench Questions 3, 4, and 5?
4. What type of recursive method do you think would be more difficult to debug: one that uses direct recursion, or one that uses indirect recursion? Why?
5. Which repetition approach is less efficient: a loop or a recursive method? Why?
6. When recursion is used to solve a problem, why must the recursive method call itself to solve a smaller version of the original problem?
7. How is a problem usually reduced with a recursive method?

# Programming Challenges

## 1. Recursive Multiplication

Write a recursive function that accepts two arguments into the parameters  $x$  and  $y$ . The function should return the value of  $x$  times  $y$ . Remember, multiplication can be performed as repeated addition:

$$7 * 4 = 4 + 4 + 4 + 4 + 4 + 4 + 4$$

## 2. `isMember` Method

Write a recursive boolean method named `isMember`. The method should accept two arguments: an array and a value. The method should return `true` if the value is found in the array, or `false` if the value is not found in the array. Demonstrate the method in a program.

## 3. String Reverser

Write a recursive method that accepts a string as its argument and prints the string in reverse order. Demonstrate the method in a program.

## 4. `maxElement` Method

Write a method named `maxElement` that returns the largest value in an array that is passed as an argument. The method should use recursion to find the largest element. Demonstrate the method in a program.

## 5. Palindrome Detector

A palindrome is any word, phrase, or sentence that reads the same forward and backwards. Here are some well-known palindromes:

- Able was I, ere I saw Elba
- A man, a plan, a canal, Panama

- Desserts, I stressed
- Kayak

Write a boolean method that uses recursion to determine whether a String argument is a palindrome. The method should return true if the argument reads the same forward and backwards. Demonstrate the method in a program.

## 6. Character Counter

Write a method that uses recursion to count the number of times a specific character occurs in an array of characters. Demonstrate the method in a program.

## 7. Recursive Power Method

Write a method that uses recursion to raise a number to a power. The method should accept two arguments: the number to be raised, and the exponent. Assume the exponent is a nonnegative integer. Demonstrate the method in a program.



### VideoNote The Recursive Power Problem

## 8. Sum of Numbers

Write a method that accepts an integer argument and returns the sum of all the integers from 1 up to the number passed as an argument. For example, if 50 is passed as an argument, the method will return the sum of 1, 2, 3, 4, . . . 50. Use recursion to calculate the sum. Demonstrate the method in a program.

## 9. Ackermann's Function

Ackermann's function is a recursive mathematical algorithm that can be used to test how well a computer performs recursion. Write a method `ackermann(m, n)` that solves Ackermann's function. Use the following

logic in your method:

```
If m = 0, then return n + 1
If n = 0, then return ackermann(m - 1, 1)
Otherwise, return ackermann(m - 1, ackermann(m, n - 1))
```

Test your method in a program that displays the return values of the following method calls:

```
ackermann(0, 0) ackermann(0, 1) ackermann(1, 1) ackermann(
ackermann(1, 3) ackermann(2, 2) ackermann(3, 2)
```

## 10. Recursive Population Class

In Programming Challenge 6 of [Chapter 5](#), you wrote a population class that predicts the size of a population of organisms after a number of days. Modify the class so it uses a recursive method instead of a loop to calculate the number of organisms.

# **Chapter 15 Databases**

## **Topics**

1. [15.1 Introduction to Database Management Systems](#)
2. [15.2 Tables, Rows, and Columns](#)
3. [15.3 Introduction to the SQL SELECT Statement](#)
4. [15.4 Inserting Rows](#)
5. [15.5 Updating and Deleting Existing Rows](#)
6. [15.6 Creating and Deleting Tables](#)
7. [15.7 Creating a New Database with JDBC](#)
8. [15.8 Scrollable Result Sets](#)
9. [15.9 Result Set Metadata](#)
10. [15.10 Relational Data](#)
11. [15.11 Advanced Topics](#)
12. [15.12 Common Errors to Avoid](#)

# 15.1 Introduction to Database Management Systems

## Concept:

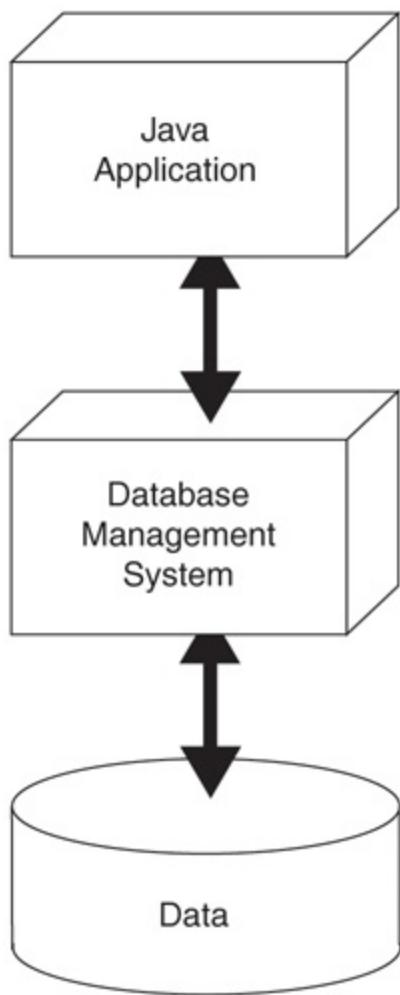
A database management system (DBMS) is software that manages large collections of data.

If an application needs to store only a small amount of data, text and binary files work well. These types of files, however, are not practical when a large amount of data must be stored and manipulated. Many businesses keep hundreds of thousands, or even millions of data items in files. When a text or binary file contains this much data, simple operations such as searching, inserting, and deleting become cumbersome and inefficient.

When developing applications that work with an extensive amount of data, most developers prefer to use a *database management system*, or *DBMS*. A DBMS is software that is specifically designed to store, retrieve, and manipulate large amounts of data in an organized and efficient manner. Once the data is stored using the database management system, applications may be written in Java or other languages to communicate with the DBMS. Rather than retrieving or manipulating the data directly, applications can send instructions to the DBMS. The DBMS carries out those instructions and sends the results back to the application. [Figure 15-1](#) illustrates this.

**Figure 15-1 A Java application interacts with a DBMS, which**

# manipulates data



[Figure 15-1 Full Alternative Text](#)

Although [Figure 15-1](#) is simplified, it illustrates the layered nature of an application that works with a database management system. The topmost layer of software, which in this case is written in Java, interacts with the user. It also sends instructions to the next layer of software, the DBMS. The DBMS works directly with the data and sends the results of operations back to the application.

For example, suppose a company keeps all of its product records in a database. The company has a Java application that allows the user to look up information on any product by entering its product ID number. The Java

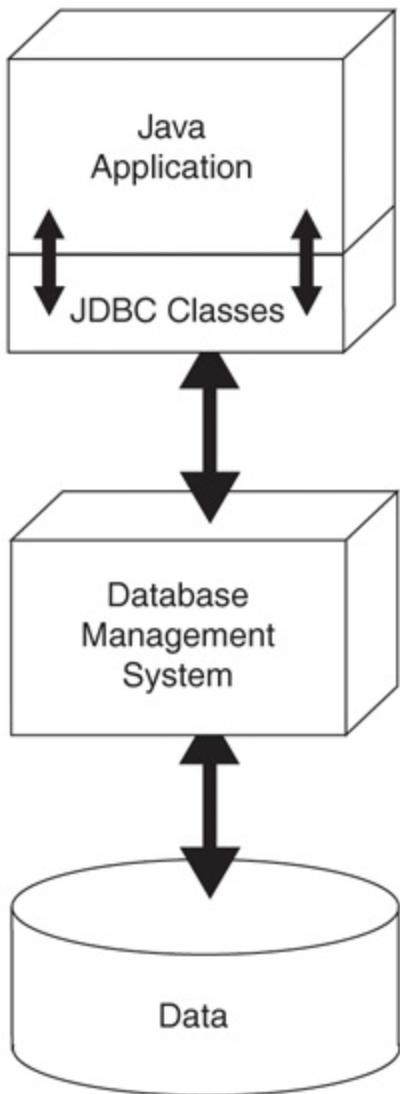
application instructs the DBMS to retrieve the record for the product with the specified product ID number. The DBMS retrieves the product record and sends the data back to the Java application. The Java application, then displays the data to the user.

The advantage of this layered approach to software development is that the Java application does not need know about the physical structure of the data. It only needs to know how to interact with the DBMS. The DBMS handles the actual reading, writing, and searching of data.

## JDBC

[Figure 15-1](#) gives a simple illustration of a Java application communicating with a DBMS. The technology that makes this communication possible is known as *JDBC*, which stands for *Java Database Connectivity*. The Java API contains numerous JDBC classes that allow your Java applications to interact with a DBMS. This is illustrated in [Figure 15-2](#).

**Figure 15-2 A Java application uses the JDBC classes to interact with a DBMS**



[Figure 15-2 Full Alternative Text](#)

## SQL

SQL, which stands for *Structured Query Language*, is a standard language for working with database management systems. It was originally developed by IBM in the 1970s. Since then, SQL has been adopted by most database software vendors as the language of choice for interacting with their DBMS.

SQL consists of several key words. You use the key words to construct statements, which are also known as *queries*. These statements, or queries,

are submitted to the DBMS and are instructions for the DBMS to carry out operations on its data. When a Java application interacts with a DBMS, the Java application must construct SQL statements as strings then use an API method to pass those strings to the DBMS. In this chapter, you will learn how to construct simple SQL statements, then pass them to a DBMS using API method call.



## Note:

Although SQL is a language, you don't use it to write applications. It is intended only as a standard means of interacting with a DBMS. You still need a general programming language, such as Java, to write an application for the ordinary user.

# Using a DBMS

In order to use JDBC to work with a database, you will need a DBMS installed on your system, or available to you in a school lab environment. There are many commercial DBMS packages available. Oracle, Microsoft SQL Server, DB2, and MySQL are just a few of the popular ones. In your school's lab, you may already have access to one of these, or perhaps another DBMS.

# Java DB

If you do not have access to a DBMS in a school lab, you can use Java DB, which is included with the JDK. Java DB is an open source distribution of Apache Derby, a pure Java DBMS. It is automatically installed on your system when you install the JDK version 7 or higher. All of the examples in this chapter were created with Java DB. If you wish to use Java DB, see Appendix I—Configuring Java DB, available for download at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).

# Creating the CoffeeDB Database

In this chapter, we will use a database named `CoffeeDB` as our example. The `CoffeeDB` database is used in the business operations of The Midnight Roastery, a small coffee roasting company. After you have installed the Java DB DBMS, perform the following steps to create the `CoffeeDB` database:

1. Make sure you have downloaded Student Source Code files from [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis).
2. In this chapter's source code files, locate a program named `CreateCoffeeDB.java`.
3. Compile and execute the `CreateCoffeeDB.java` program. If Java DB is properly installed, this program will create the `CoffeeDB` database on your system.



## Note:

If you are in a school lab environment using a DBMS other than Java DB, consult with your instructor on how to modify the program to work with your specific DBMS.

## Connecting to the CoffeeDB Database

After installing Java DB and creating the `CoffeeDB` database, you should attempt to connect to the database with a Java program. A program can call the static JDBC method `DriverManager.getConnection` to get a connection to a database. There are overloaded versions of this method, but the simplest one has the following general format:



## VideoNote Connecting to a Database

```
DriverManager.getConnection(DatabaseURL);
```

The method returns a reference to a `Connection` object, which we will discuss in a moment. In the general format, `DatabaseURL` is a string known as a *database URL*. URL stands for Uniform Resource Locator. A database URL lists the protocol that should be used to access the database, the name of the database, and potentially other items. A simple database URL has the following general format:

*protocol:subprotocol:databaseName*

In this very simple general format, three items are listed, separated by colons: `protocol`, `subprotocol`, and `databaseName`. Let's take a closer look at each one:

- *protocol* is the database protocol. When using JDBC, the protocol will always be `jdbc`.
- The value for *subprotocol* will be dependent upon the particular type of DBMS to which you are connecting. If you are using Java DB, the subprotocol is `derby`.
- *databaseName* is the name of the database you are connecting to.

If we are using Java DB, the URL for the `CoffeeDB` database is:

`jdbc:derby:CoffeeDB`

The `DriverManager.getConnection` method searches for and loads a JDBC driver that is compatible with the database specified by the URL. A *JDBC driver* is a Java class that is designed to communicate with a specific DBMS. Each DBMS usually comes with its own JDBC driver. Typically, when you install a DBMS, you also update your system's `CLASSPATH` variable to include the JDBC driver's location. This will enable the JVM to find the driver class when you call the `DriverManager.getConnection` method.

When the `DriverManager.getConnection` method finds a compatible driver, it returns a `Connection` object. `Connection` is an interface in the `java.sql` package. You will need to use this `Connection` object to perform various tasks with the database, so save the reference in a variable. Here is an example of code that we can use in a Java application to get a connection to the `CoffeeDB` database using Java DB:

```
final String DB_URL = "jdbc:derby:CoffeeDB";
Connection conn = DriverManager.getConnection(DB_URL);
```

In the second statement shown, we call the `DriverManager.getConnection` method, passing the URL for the `CoffeeDB` database. The method returns a reference to a `Connection` object, which we assign to the `conn` variable. If the `DriverManager.getConnection` method fails to load an appropriate driver for the specified database, it will throw an `SQLException`.

Before going any further, compile and execute the `TestConnection.java` program shown in [Code Listing 15-1](#). It demonstrates what we've covered so far. (This program assumes Java DB has been installed, and the `CoffeeDB` database has been created.)

## Code Listing 15-1 (`TestConnection.java`)

```
1 import java.sql.*; // Needed for JDBC classes
2
3 /**
4 * This program demonstrates how to connect to
5 * a Java DB database using JDBC.
6 */
7
8 public class TestConnection
9 {
10 public static void main(String[] args)
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
```

```
16 try
17 {
18 // Create a connection to the database.
19 Connection conn = DriverManager.getConnection(DB_URL);
20 System.out.println("Connection created to CoffeeDB.");
21
22 // Close the connection.
23 conn.close();
24 System.out.println("Connection closed.");
25 }
26 catch(Exception ex)
27 {
28 System.out.println("ERROR: " + ex.getMessage());
29 }
30 }
31 }
```

## Program Output

```
Connection created to CoffeeDB.
Connection closed.
```

Notice line 1 imports all of the classes in the `java.sql` package. This package contains many of the necessary JDBC classes. Line 14 creates a string constant containing the URL for the `CoffeeDB` database.

JDBC methods throw an `SQLException` if they encounter a problem with a database. For that reason, we use a `try-catch` statement to handle any such exceptions. Let's take a closer look at the statements inside the `try` block:

- Line 19 does the following:
  - It declares a `Connection` variable named `conn`.
  - It calls the `DriverManager.getConnection` method to get a connection to the `CoffeeDB` database.
  - The `DriverManager.getConnection` method returns a reference to a `Connection` object. The reference is assigned to the `conn` variable.
- Line 20 displays a message indicating a connection was created.

- Line 23 calls the connection object's close method, which simply closes the database connection.
- Line 24 displays a message indicating that the connection is closed.

If a connection cannot be created in line 19, or the connection cannot be closed in line 23, an exception will be thrown. The catch clause in line 26 will handle the exception, and line 28 will display the exception object's default error message.

## Connecting to a Password-Protected Database

If the database that you are connecting to requires a user name and a password, you can use the following form of the `DriverManager.getConnection` method:

```
DriverManager.getConnection(DatabaseURL, Username, Password);
```

In this general format, `Username` is a string containing a valid username, and `Password` is a string containing the password.



### Checkpoint

1. 15.1 Why do most businesses use a DBMS to store their data instead of creating their own text files or binary files to hold the data?
2. 15.2 When a Java application uses a DBMS to store and manipulate data, why doesn't the Java application need to know specific details about the physical structure of the data?
3. 15.3 What is the technology that makes it possible for a Java application to communicate with a DBMS?

4. 15.4 What is the standard language for working with database management systems?
5. 15.5 What is a database URL?
6. 15.6 Suppose you have a Java DB database on your system named `InventoryDB`. What database URL would you use in a Java program to get a connection to the database?
7. 15.7 What static JDBC method do you call to get a connection to a database?

# 15.2 Tables, Rows, and Columns

## Concept:

Data stored in a database is organized into tables, rows, and columns.

A database management system stores data in a *database*. Your first step in learning to use a DBMS is to learn how data is organized inside a database. The data stored in a database is organized into one or more tables. Each *table* holds a collection of related data. The data stored in a table is then organized into rows and columns. A row is a complete set of information about a single item. The data stored in a row is divided into columns. Each column is an individual piece of information about the item.

The CoffeeDB database has a table named `Coffee`, which holds records for all of the different coffees sold by the company. [Table 15-1](#) shows the contents of the table.

**Table 15-1 The Coffee database table**

| Description           | ProdNum | Price |
|-----------------------|---------|-------|
| Bolivian Dark         | 14-001  | 8.95  |
| Bolivian Medium       | 14-002  | 8.95  |
| Brazilian Dark        | 15-001  | 7.95  |
| Brazilian Medium      | 15-002  | 7.95  |
| Brazilian Decaf       | 15-003  | 8.55  |
| Central American Dark | 16-001  | 9.95  |

|                         |        |       |
|-------------------------|--------|-------|
| Central American Medium | 16-002 | 9.95  |
| Sumatra Dark            | 17-001 | 7.95  |
| Sumatra Decaf           | 17-002 | 8.95  |
| Sumatra Medium          | 17-003 | 7.95  |
| Sumatra Organic Dark    | 17-004 | 11.95 |
| Kona Medium             | 18-001 | 18.45 |
| Kona Dark               | 18-002 | 18.45 |
| French Roast Dark       | 19-001 | 9.65  |
| Galapagos Medium        | 20-001 | 6.85  |
| Guatemalan Dark         | 21-001 | 9.95  |
| Guatemalan Decaf        | 21-002 | 10.45 |
| Guatemalan Medium       | 21-003 | 9.95  |

As you can see, the table has 18 rows. Each row holds data about a type of coffee. The rows are divided into three columns. The first column is named `Description`, and holds the description of a type of coffee. The second column is named `ProdNum`, and holds a coffee's product number. The third column is named `Price`, and holds a coffee's price per pound. As illustrated in [Figure 15-3](#), the third row in the table holds the following data:

- Description: Brazilian Dark
- Product Number: 15-001
- Price: 7.95

## Figure 15-3 The Coffee database table

This row contains data about a single item.  
 Description: Brazilian Dark  
 Product Number: 15-001  
 Price: 7.95

| Description             | ProdNum | Price |
|-------------------------|---------|-------|
| Bolivian Dark           | 14-001  | 8.95  |
| Bolivian Medium         | 14-002  | 8.95  |
| Brazilian Dark          | 15-001  | 7.95  |
| Brazilian Medium        | 15-002  | 7.95  |
| Brazilian Decaf         | 15-003  | 8.55  |
| Central American Dark   | 16-001  | 9.95  |
| Central American Medium | 16-002  | 9.95  |
| Sumatra Dark            | 17-001  | 7.95  |
| Sumatra Decaf           | 17-002  | 8.95  |
| Sumatra Medium          | 17-003  | 7.95  |
| Sumatra Organic Dark    | 17-004  | 11.95 |
| Kona Medium             | 18-001  | 18.45 |
| Kona Dark               | 18-002  | 18.45 |
| French Roast Dark       | 19-001  | 9.65  |
| Galapagos Medium        | 20-001  | 6.85  |
| Guatemalan Dark         | 21-001  | 9.95  |
| Guatemalan Decaf        | 21-002  | 10.45 |
| Guatemalan Medium       | 21-003  | 9.95  |

Description Column      ProdNum Column      Price Column

[Figure 15-3 Full Alternative Text](#)

## Column Data Types

The columns in a database table are assigned a data type. Notice the `Description` and `ProdNum` columns in the `Coffee` table hold strings, and the `Price` column holds floating-point numbers. The data types of the columns are not Java data types, however. Instead, they are SQL data types. [Table 15-2](#) lists a few of the standard SQL data types, and shows the Java data type with which each is generally compatible.

# Table 15-2 A few of the SQL data types

| SQL Data Type                             | Description                                                                                       | Corresponding Java Data Type |
|-------------------------------------------|---------------------------------------------------------------------------------------------------|------------------------------|
| INTEGER or INT                            | An integer number                                                                                 | int                          |
| CHARACTER( <i>n</i> ) or CHAR( <i>n</i> ) | A fixed-length string with a length of <i>n</i> characters                                        | String                       |
| VARCHAR( <i>n</i> )                       | A variable-length string with a maximum length of <i>n</i> characters.                            | String                       |
| REAL                                      | A single-precision floating point number                                                          | float                        |
| DOUBLE                                    | A double-precision floating point number                                                          | double                       |
| DECIMAL( <i>t</i> , <i>d</i> )            | A decimal value with <i>t</i> total digits and <i>d</i> digits appearing after the decimal point. | java.math.BigDecimal         |
| DATE                                      | A date                                                                                            | java.sql.Date                |

There are many other standard data types in SQL. When the `Coffee` table was created, the following data types were used for the columns:

- The data type for the `Description` column is `CHAR(25)`. This means that each value in the `Description` column is a string with a fixed length of 25 characters, compatible with the `String` type in Java.
- The data type for the `ProdNum` column is `CHAR(10)`. This means that each value in the `ProdNum` column is a string with a fixed length of 10 characters, compatible with the `String` type in Java.
- The data type for the `Price` column is `DOUBLE`. This means that each

value in the Price column is a double-precision floating-point number, compatible with the double data type in Java.

# Primary Keys

Most database tables have a *primary key*, which is a column that can be used to identify a specific row in a table. The column designated as the primary key holds a unique value for each row. If you try to store duplicate data in the primary key column, an error will occur.

In the Coffee table, the ProdNum column is the primary key because it holds a unique product number for each type of coffee. Here are some other examples:

- Suppose a table stores employee data, and one of the columns holds employee ID numbers. Because each employee's ID number is unique, this column can be used as the primary key.
- Suppose a table stores data about a cell phone company's inventory of phones, and one of the columns holds cell phone serial numbers. Because each phone's serial number is unique, this column can be used as the primary key.
- Suppose a table stores invoice data, and one of the columns holds invoice numbers. Each invoice has a unique invoice number, so this column can be used as a primary key.



## Note:

It is possible for a table's primary key to be the combination of several columns in the table.



## Checkpoint

1. 15.8 Describe how the data that is stored in a table is organized.
2. 15.9 What is a primary key?
3. 15.10 What Java data types correspond with the following SQL types?
  - INTEGER
  - INT
  - REAL
  - CHAR
  - CHARACTER
  - VARCHAR
  - DOUBLE

# 15.3 Introduction to the SQL SELECT Statement

## Concept:

The `SELECT` statement is used in SQL to retrieve data from a database.

The first SQL statement we will discuss is the `SELECT` statement. You use the *SELECT statement* to retrieve the rows in a table. As its name implies, the `SELECT` statement allows you to select specific rows. We will start with a very simple form of the statement, as shown here:



**VideoNote** The SQL SELECT Statement

```
SELECT Columns FROM Table
```

In the general form, *Columns* is one or more column names, and *Table* is a table name. Here is an example `SELECT` statement that we might execute on the `CoffeeDB` database:

```
SELECT Description FROM Coffee
```

This statement will retrieve the `Description` column for every row in the `Coffee` table. [Figure 15-4](#) shows an example of the results.

## Figure 15-4 Description column

| Description             |
|-------------------------|
| Bolivian Dark           |
| Bolivian Medium         |
| Brazilian Dark          |
| Brazilian Medium        |
| Brazilian Decaf         |
| Central American Dark   |
| Central American Medium |
| Sumatra Dark            |
| Sumatra Decaf           |
| Sumatra Medium          |
| Sumatra Organic Dark    |
| Kona Medium             |
| Kona Dark               |
| French Roast Dark       |
| Galapagos Medium        |
| Guatemalan Dark         |
| Guatemalan Decaf        |
| Guatemalan Medium       |

[Figure 15-4 Full Alternative Text](#)

[Figure 15-4](#) shows the results of a SELECT statement, but what happens to these results? In a Java program, the results of a SELECT statement are returned to the program in a ResultSet object. A *ResultSet object* is simply an object that contains the results of an SQL statement. The process of sending an SQL statement to a DBMS can be summarized in the following steps:

1. Get a connection to the database.
2. Pass a string containing an SQL statement to the DBMS. If the SQL statement has results to send back, a ResultSet object will be returned to the program.
3. Process the contents of the ResultSet object, if one has been returned to the program.

4. When finished working with the database, close the connection.

In [Code Listing 15-1](#), you saw an example of how to perform step 1 (get a connection to the database) and step 4 (close the connection). Let's look at the details of how an SQL statement is sent to the DBMS and its results are processed in steps 2 and 3.

## Passing an SQL Statement to the DBMS

Once you have gotten a connection to the database, you are ready to issue SQL statements to the DBMS. First, you must get a `Statement` object from the `Connection` object, using its `createStatement` method. Here is an example:

```
Statement stmt = conn.createStatement();
```

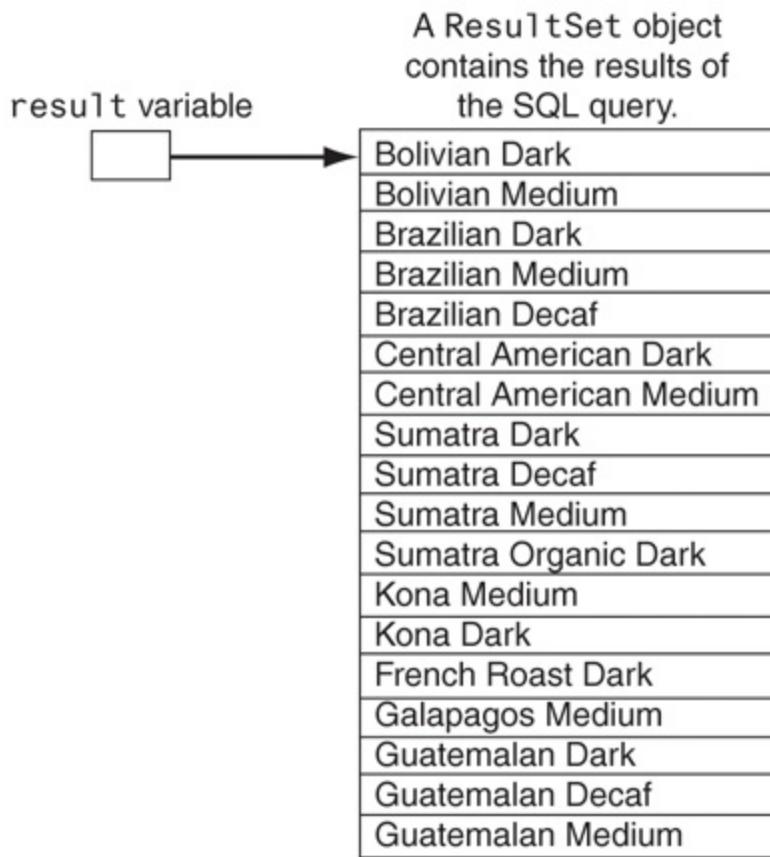
After this code executes, the `stmt` variable will reference a `Statement` object. `Statement` is an interface in the `java.sql` package. `Statement` objects have a variety of methods that can be used to execute SQL queries. To execute a `SELECT` query, you use the `executeQuery` method. The method returns a `ResultSet` object. Here is an example:

```
String sqlStatement = "SELECT Description FROM Coffee";
ResultSet result = stmt.executeQuery(sqlStatement);
```

The first statement creates a string containing an SQL query. The second statement passes this string as an argument to the `executeQuery` method. The method returns a reference to a `ResultSet` object containing the results of the query. The reference is assigned to the `result` variable. [Figure 15-5](#) illustrates how the `result` variable references the `ResultSet` object.

## Figure 15-5 A `ResultSet` object contains the results of an SQL

# query



[Figure 15-5 Full Alternative Text](#)

A ResultSet object contains a set of rows and columns. The ResultSet object in [Figure 15-5](#) has eighteen rows and one column. The rows in a ResultSet are numbered, with the first row being row 1, the second row being row 2, and so forth. The columns are also numbered, with the first column being column 1, the second column being column 2, and so forth. [Figure 15-6](#) shows the same ResultSet with the row and column numbers labeled.

## Figure 15-6 ResultSet rows and columns

|        | Column 1                |
|--------|-------------------------|
| Row 1  | Bolivian Dark           |
| Row 2  | Bolivian Medium         |
| Row 3  | Brazilian Dark          |
| Row 4  | Brazilian Medium        |
| Row 5  | Brazilian Decaf         |
| Row 6  | Central American Dark   |
| Row 7  | Central American Medium |
| Row 8  | Sumatra Dark            |
| Row 9  | Sumatra Decaf           |
| Row 10 | Sumatra Medium          |
| Row 11 | Sumatra Organic Dark    |
| Row 12 | Kona Medium             |
| Row 13 | Kona Dark               |
| Row 14 | French Roast Dark       |
| Row 15 | Galapagos Medium        |
| Row 16 | Guatemalan Dark         |
| Row 17 | Guatemalan Decaf        |
| Row 18 | Guatemalan Medium       |

[Figure 15-6 Full Alternative Text](#)

## Getting a Row from the ResultSet Object

A `ResultSet` object has an internal *cursor* that points to a specific row in the `ResultSet`. The row to which the cursor points is considered the *current row*. The cursor can be moved from row to row, and this provides you with a way to examine all of the rows in the `ResultSet`.

At first, the cursor is not pointing to a row, but is positioned just before the first row. This is illustrated in [Figure 15-7](#).

## Figure 15-7 The cursor is

# initially positioned before the first row

Initially, the cursor is positioned just before the first row in the ResultSet.

Cursor → Column 1

|        |                         |
|--------|-------------------------|
| Row 1  | Bolivian Dark           |
| Row 2  | Bolivian Medium         |
| Row 3  | Brazilian Dark          |
| Row 4  | Brazilian Medium        |
| Row 5  | Brazilian Decaf         |
| Row 6  | Central American Dark   |
| Row 7  | Central American Medium |
| Row 8  | Sumatra Dark            |
| Row 9  | Sumatra Decaf           |
| Row 10 | Sumatra Medium          |
| Row 11 | Sumatra Organic Dark    |
| Row 12 | Kona Medium             |
| Row 13 | Kona Dark               |
| Row 14 | French Roast Dark       |
| Row 15 | Galapagos Medium        |
| Row 16 | Guatemalan Dark         |
| Row 17 | Guatemalan Decaf        |
| Row 18 | Guatemalan Medium       |

[Figure 15-7 Full Alternative Text](#)

To move the cursor to the first row in a newly created ResultSet, you call the object's next method. Here is an example:

```
result.next();
```

Assuming that result references a newly created ResultSet object, this statement moves the cursor to the first row in the ResultSet. [Figure 15-8](#) shows how the cursor has moved to the first row in the ResultSet after the next method is called the first time.

# Figure 15-8 The next method moves the cursor forward

After the `ResultSet` object's `next` method is called the first time, the cursor is positioned at the first row.

|          | Column 1                      |
|----------|-------------------------------|
| Cursor → | Row 1 Bolivian Dark           |
|          | Row 2 Bolivian Medium         |
|          | Row 3 Brazilian Dark          |
|          | Row 4 Brazilian Medium        |
|          | Row 5 Brazilian Decaf         |
|          | Row 6 Central American Dark   |
|          | Row 7 Central American Medium |
|          | Row 8 Sumatra Dark            |
|          | Row 9 Sumatra Decaf           |
|          | Row 10 Sumatra Medium         |
|          | Row 11 Sumatra Organic Dark   |
|          | Row 12 Kona Medium            |
|          | Row 13 Kona Dark              |
|          | Row 14 French Roast Dark      |
|          | Row 15 Galapagos Medium       |
|          | Row 16 Guatemalan Dark        |
|          | Row 17 Guatemalan Decaf       |
|          | Row 18 Guatemalan Medium      |

## [Figure 15-8 Full Alternative Text](#)

Each time you call the `next` method, it moves the cursor to the next row in the `ResultSet`.

The `next` method returns a boolean value. It returns `true` if the cursor successfully moved to the next row. If there are no more rows, it returns `false`. The following code shows how you can move the cursor through all of the rows of a newly created `ResultSet`:

```
while (result.next())
{
```

```
// Process the current row.
}
```

There are other `ResultSet` methods for navigating the rows in a `ResultSet` object. We will look at some of them later in this chapter.

## Getting Columns in a `ResultSet` Row

You use one of the `ResultSet` object's "get" methods to retrieve the contents of a specific column in the current row. When you call one of these methods, you can pass either the column number or the column name as an argument. There are numerous "get" methods defined in the `ResultSet` interface. [Table 15-3](#) lists a few of them.

**Table 15-3 A few of the `ResultSet` methods**

| <b>ResultSet Method</b>                       | <b>Description</b>                                                                                                                                                                                                                             |
|-----------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>double getDouble(int colNumber)</code>  | Returns the double stored in the column specified by <i>colNumber</i> or <i>colName</i> . The column must hold data that is compatible with the double data type in Java. If an error occurs, the method throws an <code>SQLException</code> . |
| <code>double getDouble(String colName)</code> |                                                                                                                                                                                                                                                |
| <code>int getInt(int colNumber)</code>        | Returns the int stored in the column specified by <i>colNumber</i> or <i>colName</i> . The column must hold data that is compatible with the int data type in Java. If an error occurs, the method                                             |
| <code>int getInt(String</code>                |                                                                                                                                                                                                                                                |

*colName*)

throws an `SQLException`.

`String  
getString(int  
colNumber)`

Returns the string stored in the column specified by *colNumber* or *colName*. The column must hold data that is compatible with the `String` type in Java. If an error occurs, the method throws an `SQLException`.

`String  
getString(String  
colName)`

Recall that columns have an SQL data type. The SQL data types are not the same as the Java data types, but are compatible with them. To retrieve the contents of a column, you call the method that is designed to work with that column's data type. For example, if the column contains a string, you would use the `getString` method to retrieve its value. If the column contains an integer, you would use the `getInt` method. Likewise, if the column contains a double, you would call the `getDouble` method.

When you call one of the “get” methods, you must tell it which column in the current row you want to retrieve. These methods accept either an integer argument, which is a column number, or a `String` holding the column name.

The `ResultSet` that we have been looking at in our example has only one column: the `Description` column. The `Description` column's data type is `CHAR(25)`, which means it is a fixed-length string of 25 characters. This is compatible with the `String` type in Java. To display the contents of the `Description` column in the current row, we could use the following statement:

```
System.out.println(result.getString("Description"));
```

The `Description` column holds values that are compatible with the `String` type, so we use the `getString` method to retrieve its contents. We could also use the column number to retrieve the column contents. Here is an example:

```
System.out.println(result.getString(1));
```



## Note:

Column names in a database table are not case-sensitive. The column names DESCRIPTION, description, and Description are all the same.

Let's look at a complete program that demonstrates what we have covered so far. [Code Listing 15-2](#) displays the Description column from all of the rows in the CoffeeDB database.

## Code Listing 15-2 (ShowCoffeeDescriptions.java)

```
1 import java.sql.*; // Needed for JDBC classes
2
3 /**
4 * This program demonstrates how to issue an SQL
5 * SELECT statement to a database using JDBC.
6 */
7
8 public class ShowCoffeeDescriptions
9 {
10 public static void main(String[] args)
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
16 try
17 {
18 // Create a connection to the database.
19 Connection conn = DriverManager.getConnection(DB_URL);
20
21 // Create a Statement object.
22 Statement stmt = conn.createStatement();
23
24 // Create a string with a SELECT statement.
25 String sqlStatement = "SELECT Description FROM Coffee";
26
27 // Send the statement to the DBMS.
```

```
28 ResultSet result = stmt.executeQuery(sqlStatement);
29
30 // Display a header for the listing.
31 System.out.println("Coffees Found in the Database");
32 System.out.println("-----");
33
34 // Display the contents of the result set.
35 // The result set will have three columns.
36 while (result.next())
37 {
38 System.out.println(result.getString("Description"));
39 }
40
41 // Close the connection.
42 conn.close();
43 }
44 catch(Exception ex)
45 {
46 System.out.println("ERROR: " + ex.getMessage());
47 }
48 }
49 }
```

## Program Output

Coffees Found in the Database

-----  
Bolivian Dark  
Bolivian Medium  
Brazilian Dark  
Brazilian Medium  
Brazilian Decaf  
Central American Dark  
Central American Medium  
Sumatra Dark  
Sumatra Decaf  
Sumatra Medium  
Sumatra Organic Dark  
Kona Medium  
Kona Dark  
French Roast Dark  
Galapagos Medium  
Guatemalan Dark  
Guatemalan Decaf  
Guatemalan Medium

Let's take a closer look at the code. Line 14 declares a string constant,

initialized with the URL for the `CoffeeDB` database. The statements that access the database are written inside the `try` block that appears in lines 18 through 43. Line 19 gets a connection to the database. After line 19 executes, the `conn` variable will reference a `Connection` object that can be used to access the database.

At this point in the program, we have a connection to the `CoffeeDB` database, but we are not ready to send a `SELECT` statement to the database. In order to send a `SELECT` statement to the database, we must have a `Statement` object. Line 22 calls the `Connection` object's `createStatement` method, which returns a reference to a `Statement` object. The reference is assigned to the `stmt` variable.

Line 25 declares a `String` variable named `sqlStatement`, initialized with the string `"SELECT Description FROM Coffee"`. This is the SQL statement we want to submit to the database. Line 28 passes this string as an argument to the `Statement` object's `executeQuery` method, which executes the statement. The method returns a reference to a `ResultSet` object, which is assigned to the `resultSet` variable. The `ResultSet` object contains the results of the `SELECT` statement.

The `while` loop that appears in lines 36 through 39 displays the contents of the `ResultSet` object. It works like this:

- The `while` statement in line 36 calls the `ResultSet` object's `next` method to advance the internal cursor. If the cursor is successfully advanced, the method returns `true`, and the loop iterates. If the cursor is at the end of the `ResultSet` object's rows, the method returns `false` and the loop terminates.
- Each time the loop iterates, the `ResultSet` object's internal cursor will be positioned at a specific row. The statement in line 38 gets the value of the `Description` column and displays it.

Line 42 closes the connection to the database.

## More about the `SELECT` Statement

You can specify more than one column in a SELECT statement by separating the column names with commas. Here is an example:

```
SELECT Description, Price FROM Coffee
```

This statement will retrieve the Description column and the Price column for every row in the Coffee table. The program in [Code Listing 15-3](#) displays the coffee descriptions and their prices.

## Code Listing 15-3 (ShowDescriptionsAndPrices.java)

```
1 import java.sql.*; // Needed for JDBC classes
2
3 /**
4 * This program displays the coffee descriptions
5 * and their prices.
6 */
7
8 public class ShowDescriptionsAndPrices
9 {
10 public static void main(String[] args)
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
16 try
17 {
18 // Create a connection to the database.
19 Connection conn = DriverManager.getConnection(DB_URL);
20
21 // Create a Statement object.
22 Statement stmt = conn.createStatement();
23
24 // Create a string with a SELECT statement.
25 String sqlStatement =
26 "SELECT Description, Price FROM Coffee";
27
28 // Send the statement to the DBMS.
29 ResultSet result = stmt.executeQuery(sqlStatement);
30 }
```

```

31 // Display the contents of the result set.
32 // The result set will have three columns.
33 while (result.next())
34 {
35 System.out.printf("%25s %.2f\n",
36 result.getString("Description"),
37 result.getDouble("Price"));
38 }
39
40 // Close the connection.
41 conn.close();
42 }
43 catch(Exception ex)
44 {
45 System.out.println("ERROR: " + ex.getMessage());
46 }
47 }
48 }
```

## Program Output

|                         |       |
|-------------------------|-------|
| Bolivian Dark           | 8.95  |
| Bolivian Medium         | 8.95  |
| Brazilian Dark          | 7.95  |
| Brazilian Medium        | 7.95  |
| Brazilian Decaf         | 8.55  |
| Central American Dark   | 9.95  |
| Central American Medium | 9.95  |
| Sumatra Dark            | 7.95  |
| Sumatra Decaf           | 8.95  |
| Sumatra Medium          | 7.95  |
| Sumatra Organic Dark    | 11.95 |
| Kona Medium             | 18.45 |
| Kona Dark               | 18.45 |
| French Roast Dark       | 9.65  |
| Galapagos Medium        | 6.85  |
| Guatemalan Dark         | 9.95  |
| Guatemalan Decaf        | 10.45 |
| Guatemalan Medium       | 9.95  |

The program in [Code Listing 15-3](#) is very similar to [Code Listing 15-2](#). The differences between the two programs are summarized here:

- Lines 25 and 26 initialize the `sqlStatement` variable with the string "`SELECT Description, Price FROM Coffee`". This is a SELECT

statement that will retrieve the Description and Price columns from the database table.

- Inside the while loop, in lines 33 through 38, we call result.getString to get the current row's Description column, and we call result.getDouble to get the current row's Price column. These items are displayed with the System.out.printf method.

If you wish to retrieve every column, you can use the \* character instead of listing column names. Here is an example:

```
SELECT * FROM Coffee
```

This statement will retrieve every column for every row in the Coffee table. The program in [Code Listing 15-4](#) displays all the columns in the Coffee table.

## Code Listing 15-4 (ShowCoffeeData.java)

```
1 import java.sql.*; // Needed for JDBC classes
2
3 /**
4 * This program displays all of the columns in the
5 * Coffee table of the CoffeeDB database.
6 */
7
8 public class ShowCoffeeData
9 {
10 public static void main(String[] args)
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
16 try
17 {
18 // Create a connection to the database.
19 Connection conn = DriverManager.getConnection(DB_URL);
20
```

```

21 // Create a Statement object.
22 Statement stmt = conn.createStatement();
23
24 // Create a string with a SELECT statement.
25 String sqlStatement = "SELECT * FROM Coffee";
26
27 // Send the statement to the DBMS.
28 ResultSet result = stmt.executeQuery(sqlStatement);
29
30 // Display the contents of the result set.
31 // The result set will have three columns.
32 while (result.next())
33 {
34 System.out.printf("%25s %6s %5.2f\n",
35 result.getString("Description"),
36 result.getString("ProdNum"),
37 result.getDouble("Price"));
38 }
39
40 // Close the connection.
41 conn.close();
42 }
43 catch(Exception ex)
44 {
45 System.out.println("ERROR: " + ex.getMessage());
46 }
47 }
48 }
```

## Program Output

|                         |        |       |
|-------------------------|--------|-------|
| Bolivian Dark           | 14-001 | 8.95  |
| Bolivian Medium         | 14-002 | 8.95  |
| Brazilian Dark          | 15-001 | 7.95  |
| Brazilian Medium        | 15-002 | 7.95  |
| Brazilian Decaf         | 15-003 | 8.55  |
| Central American Dark   | 16-001 | 9.95  |
| Central American Medium | 16-002 | 9.95  |
| Sumatra Dark            | 17-001 | 7.95  |
| Sumatra Decaf           | 17-002 | 8.95  |
| Sumatra Medium          | 17-003 | 7.95  |
| Sumatra Organic Dark    | 17-004 | 11.95 |
| Kona Medium             | 18-001 | 18.45 |
| Kona Dark               | 18-002 | 18.45 |
| French Roast Dark       | 19-001 | 9.65  |
| Galapagos Medium        | 20-001 | 6.85  |
| Guatemalan Dark         | 21-001 | 9.95  |

|                   |        |         |
|-------------------|--------|---------|
| Guatemalan Decaf  | 21-002 | 10 . 45 |
| Guatemalan Medium | 21-003 | 9 . 95  |

SQL statements are free-form, which means tabs, newlines, and spaces between the key words are ignored. For example, the statement

```
SELECT * FROM Coffee
```

works the same as:

```
SELECT
```

```
*
```

```
FROM
```

```
Coffee
```

In addition, SQL key words and table names are case-insensitive. The previous statement could be written as:

```
select * from coffee
```

## Specifying a Search Criteria with the WHERE Clause

Occasionally, you might want to retrieve every row in a table. For example, if you wanted a list of all the coffees in the `Coffee` table, the previous `SELECT` statement would give it to you. Normally, however, you want to narrow the list down to only a few selected rows in the table. That's where the `WHERE` clause comes in. The `WHERE` clause can be used with the `SELECT` statement to specify a search criteria. When you use the `WHERE` clause, only the rows that meet the search criteria will be returned in the result set. The general format of a `SELECT` statement with a `WHERE` clause is:

```
SELECT Columns FROM Table WHERE Criteria
```

In the general format, `Criteria` is a conditional expression. Here is an

example of a SELECT statement that uses the WHERE clause:

```
SELECT * FROM Coffee WHERE Price > 12.00
```

The first part of the statement, `SELECT * FROM Coffee`, specifies that we want to see every column. The WHERE clause specifies that we only want the rows in which the contents of the Price column is greater than 12.00. [Figure 15-9](#) shows the results of this statement. Notice only two coffees meet this search criterion.

## Figure 15-9 Rows where Price is greater than 12.00

| Description | ProdNum | Price |
|-------------|---------|-------|
| Kona Medium | 18-001  | 18.45 |
| Kona Dark   | 18-002  | 18.45 |

[Figure 15-9 Full Alternative Text](#)

Standard SQL supports the relational operators listed in [Table 15-4](#) for writing conditional expressions in a WHERE clause.

## Table 15-4 SQL relational operators

| Operator | Meaning                  |
|----------|--------------------------|
| >        | Greater than             |
| <        | Less than                |
| >=       | Greater than or equal to |
| <=       | Less than or equal to    |
| =        | Equal to                 |

<> Not equal to

Notice the equal-to and not-equal-to operators in SQL are different from those in Java. The equal-to operator is one equal sign, not two equal signs. The not-equal-to operator is <>.

Let's look at a few more examples of the SELECT statement. The following statement could be used to retrieve the product numbers and prices of all the coffees that are priced at 7.95:

```
SELECT ProdNum, Price FROM Coffee WHERE Price = 7.95
```

The results of this statement are shown in [Figure 15-10](#).

## Figure 15-10 Results of SQL statement

| ProdNum | Price |
|---------|-------|
| 15-001  | 7.95  |
| 15-002  | 7.95  |
| 17-001  | 7.95  |
| 17-003  | 7.95  |

[Figure 15-10 Full Alternative Text](#)

The following SELECT statement retrieves all of the columns for the row where the description is "French Roast Dark". The results returned from this statement are shown in [Figure 15-11](#).

## Figure 15-11 Results of SQL statement

| Description       | ProdNum | Price |
|-------------------|---------|-------|
| French Roast Dark | 19-001  | 9.65  |

[Figure 15-11 Full Alternative Text](#)

```
SELECT * FROM Coffee WHERE Description = 'French Roast Dark'
```

If you look carefully at the previous statement, you will notice another difference between SQL syntax and Java syntax. In SQL, string literals are enclosed in single quotes, not double quotes.

## Tip:

If you need to include a single quote as part of a string, simply write two single quotes in its place. For example, suppose you wanted to search the Coffee table for a coffee named Joe's Special Blend. You could use the following statement:

```
SELECT * FROM Coffee WHERE Description = 'Joe''s Special Blend'
```

Let's look at an example program that uses a WHERE clause in a SELECT statement. The program in [Code Listing 15-5](#) lets the user enter a minimum price, then search the Coffee table for rows where the Price column is greater than or equal to the specified price.

## Code Listing 15-5 (CoffeeMinPrice.java)

```

1 import java.util.Scanner;
2 import java.sql.*;
3
4 /**
5 * This program lets the user search for coffees
6 * priced at a minimum value.
7 */
8
```

```
9 public class CoffeeMinPrice
10 {
11 public static void main(String[] args)
12 {
13 double minPrice; // To hold the minimum price
14 int coffeeCount = 0; // To count coffees that qualify
15
16 // Create a named constant for the URL.
17 // NOTE: This value is specific for Java DB.
18 final String DB_URL = "jdbc:derby:CoffeeDB";
19
20 // Create a Scanner object for keyboard input.
21 Scanner keyboard = new Scanner(System.in);
22
23 // Get the minimum price from the user.
24 System.out.print("Enter the minimum price: ");
25 minPrice = keyboard.nextDouble();
26
27 try
28 {
29 // Create a connection to the database.
30 Connection conn = DriverManager.getConnection(DB_URL);
31
32 // Create a Statement object.
33 Statement stmt = conn.createStatement();
34
35 // Create a string containing a SELECT statement.
36 // Note that we are incorporating the user's input
37 // into the string.
38 String sqlStatement =
39 "SELECT * FROM Coffee WHERE Price >= " +
40 Double.toString(minPrice);
41
42 // Send the statement to the DBMS.
43 ResultSet result = stmt.executeQuery(sqlStatement);
44
45 // Display the contents of the result set.
46 // The result set will have three columns.
47 while (result.next())
48 {
49 // Display a row from the result set.
50 System.out.printf("%25s %6s %5.2f\n",
51 result.getString("Description"),
52 result.getString("ProdNum"),
53 result.getDouble("Price"));
54
55 // Increment the counter.
```

```

56 coffeeCount++;
57 }
58
59 // Display the number of qualifying coffees.
60 System.out.println(coffeeCount + " coffees found.");
61
62 // Close the connection.
63 conn.close();
64 }
65 catch(Exception ex)
66 {
67 System.out.println("ERROR: " + ex.getMessage());
68 }
69 }
70 }

```

## Program Output

Enter the minimum price: **12.00**   
 Kona Medium 18-001 18.45  
 Kona Dark 18-002 18.45  
 2 coffees found.

## Program Output

Enter the minimum price: **10.00**   
 Sumatra Organic Dark 17-004 11.95  
 Kona Medium 18-001 18.45  
 Kona Dark 18-002 18.45  
 Guatemalan Decaf 21-002 10.45  
 4 coffees found.

## Program Output

Enter the minimum price: **20.00**   
 0 coffees found.

There are a few things in [Code Listing 15-5](#) that deserve some explanation. In lines 24 and 25, the program prompts the user to enter a minimum price, and the user's input is assigned to the double variable `minPrice`. Then, notice in lines 38 through 40, the `minPrice` variable is converted to a string and concatenated onto the string containing the `SELECT` statement. When the program runs, if the user enters 10.00, the `SELECT` statement that is created in

lines 38 through 40 will be:

```
SELECT * FROM Coffee WHERE Price >= 10.00
```

Or, if the user enters 12.00, the SELECT statement that is created in lines 38 through 40 will be:

```
SELECT * FROM Coffee WHERE Price >= 12.00
```

Programs commonly need to use techniques such as this to create SQL statements that incorporate user input.

## String Comparisons in a SELECT Statement

String comparisons in SQL are case-sensitive. If you ran the following statement against the coffeeDB database, you would not get any results:

```
SELECT * FROM Coffee WHERE Description = 'french roast dark'
```

However, you can use the UPPER( ) function to convert a string to uppercase. Here is an example:

```
SELECT * FROM Coffee WHERE UPPER(Description) = 'FRENCH ROAST DAR
```

This statement will return the same results as shown in [Figure 15-11](#). SQL also provides a LOWER( ) function, which converts its argument to lowercase.

## Using the LIKE Operator

Sometimes searching for an exact string will not yield the results you want. For example, suppose we want a list of all the decaf coffees in the Coffee table. The following statement will not work. Can you see why?

```
SELECT * FROM Coffee WHERE Description = 'Decaf'
```

This statement will search for rows where the Description field is equal to the string “Decaf”. Unfortunately, it will find none. If you glance back at [Table 15-1](#), you will see that none of the rows in the Coffee table have a Description column that is equal to “Decaf”. You will see, however, that the word “Decaf” does appear in the Description column of some of the rows. For example, in one row you will find “Brazilian Decaf”. In another row you will find “Sumatra Decaf”. In yet another row you will find “Guatemalan Decaf”. In addition to the word “Decaf”, each of these contains other characters.

In order to find all of the decaf coffees, we need to search for rows where “Decaf” appears as a substring in the Description column. You can perform such a search using the LIKE operator. Here is an example of how to use it:

```
SELECT * FROM Coffee WHERE Description LIKE '%Decaf%'
```

The LIKE operator is followed by a string that contains a character pattern. In this example, the character pattern is '%Decaf%'. The % symbol is used as wildcard character. It represents any sequence of 0 or more characters. The pattern '%Decaf%' specifies that the string “Decaf” must appear with any sequence of characters before or after it. The results of this statement are shown in [Figure 15-12](#).

## Figure 15-12 Results of SQL statement

| Description      | ProdNum | Price |
|------------------|---------|-------|
| Brazilian Decaf  | 15-003  | 8.55  |
| Sumatra Decaf    | 17-002  | 8.95  |
| Guatemalan Decaf | 21-002  | 10.45 |

[Figure 15-12 Full Alternative Text](#)

Likewise, the following statement will result in all the rows where the

Description column starts with the word “Sumatra”:

```
SELECT * FROM Coffee WHERE Description LIKE 'Sumatra%'
```

The underscore character ( \_ ) is also used as a wildcard. Unlike the % character, the underscore represents a single character. For example, look at the following statement:

```
SELECT * FROM Coffee WHERE ProdNum LIKE '2_-00_'
```

This statement will result in all the rows where the ProdNum column begins with “2”, followed by any single character, followed by “-00”, followed by any single character. The results of this statement are shown in [Figure 15-13](#).

## Figure 15-13 Results of SQL statement

| Description       | ProdNum | Price |
|-------------------|---------|-------|
| Galapagos Medium  | 20-001  | 6.85  |
| Guatemalan Dark   | 21-001  | 9.95  |
| Guatemalan Decaf  | 21-002  | 10.45 |
| Guatemalan Medium | 21-003  | 9.95  |

[Figure 15-13 Full Alternative Text](#)

You can use the NOT operator to disqualify a character pattern in a search criteria. For example, suppose you want a list of all the coffees that are not decaf. The following statement will yield just those results:

```
SELECT * FROM Coffee WHERE Description NOT LIKE '%Decaf%'
```

## Using AND and OR

You can use the AND and OR logical operators to specify multiple search criteria in a WHERE clause. For example, look at the following statement:

```
SELECT * FROM Coffee WHERE Price > 10.00 AND Price < 14.00
```

The AND operator requires that both of the search criteria be true in order for a row to be qualified as a match. The only rows that will be returned from this statement are those where the Price column contains a value that is greater than 10.00 and less than 14.00. [Figure 15-14](#) shows the results of the statement.

## Figure 15-14 Results of SQL statement

| Description          | ProdNum | Price |
|----------------------|---------|-------|
| Sumatra Organic Dark | 17-004  | 11.95 |
| Guatemalan Decaf     | 21-002  | 10.45 |

[Figure 15-14 Full Alternative Text](#)

Here's an example that uses the OR operator:

```
SELECT * FROM Coffee
WHERE Description LIKE '%Dark%' OR ProdNum LIKE '16%'
```

The OR operator requires that either of the search criteria be true in order for a row to be qualified as a match. This statement searches for rows where the Description column contains the string "Dark" at any position, or where the ProdNum column starts with "16". [Figure 15-15](#) shows the results of the query.

## Figure 15-15 Results of the SQL statement

| Description             | ProdNum | Price |
|-------------------------|---------|-------|
| Bolivian Dark           | 14-001  | 8.95  |
| Brazilian Dark          | 16-001  | 7.95  |
| Central American Dark   | 16-001  | 9.95  |
| Central American Medium | 16-002  | 9.95  |
| Sumatra Dark            | 17-001  | 7.95  |
| Sumatra Organic Dark    | 17-004  | 11.95 |
| Kona Dark               | 18-002  | 18.45 |
| French Roast Dark       | 19-001  | 9.65  |
| Guatemalan Dark         | 21-001  | 9.95  |

[Figure 15-15 Full Alternative Text](#)

## Sorting the Results of a SELECT Query

If you wish to sort the results of a SELECT query, you can use the ORDER BY clause. Here is an example:

```
SELECT * FROM Coffee ORDER BY Price
```

This statement will produce a list of all the rows in the coffee table, ordered by the Price column. The list will be sorted in ascending order on the Price column, meaning that the lowest priced coffees will appear first.

Here's a SELECT statement that uses both a WHERE clause and an ORDER BY clause:

```
SELECT * FROM Coffee
WHERE Price > 9.95
ORDER BY Price
```

This statement will produce a list of all the rows in the coffee table where

the Price column contains a value greater than 9.95, listed in ascending order by price.

If you want the list sorted in descending order (from highest to lowest), use the DESC operator, as shown here:

```
SELECT * FROM Coffee
WHERE Price > 9.95
ORDER BY Price DESC
```

## Mathematical Functions

SQL provides several functions for performing calculations. For example, the AVG function calculates the average value in a particular column. Here is an example SELECT statement using the AVG function:

```
SELECT AVG(Price) FROM Coffee
```

This statement produces a single value: the average of all the values in the Price column. Because we did not use a WHERE clause, it uses all of the rows in the Coffee table in the calculation. Here is an example that calculates the average price of all the coffees having a product number that begins with “20”:

```
SELECT AVG(Price) FROM Coffee WHERE ProdNum LIKE '20%'
```

Another of the mathematical functions is SUM, which calculates the sum of a column’s values. The following statement, which is probably not very useful, calculates the sum of the values in the Price column:

```
SELECT SUM(Price) FROM Coffee
```

The MIN and MAX functions determine the minimum and maximum values found in a column. The following statement will tell us the minimum value in the Price column:

```
SELECT MIN(Price) FROM Coffee
```

The following statement will tell us the maximum value in the Price column:

```
SELECT MAX(Price) FROM Coffee
```

The COUNT function can be used to determine the number of rows in a table, as demonstrated by the following statement:

```
SELECT COUNT(*) FROM Coffee
```

The \* simply indicates you want to count entire rows. Here is another example, which tells us the number of coffees having a price greater than 9.95:

```
SELECT COUNT(*) FROM Coffee WHERE Price > 9.95
```

Queries that use math functions, such as the examples shown here, return only one value. So, when you submit such a statement to a DBMS using JDBC, the ResultSet object that is returned to the program will contain one row with one column. The program shown in [Code Listing 15-6](#) shows an example of how you can use the functions to display the MIN, MAX, and AVG functions to find the lowest, highest, and average prices in the Coffee table.

## Code Listing 15-6 (CoffeeMinPrice.java)

```
1 import java.sql.*;
2
3 /**
4 * This program demonstrates some of the SQL math functions.
5 */
6
7 public class CoffeeMath
8 {
9 public static void main(String[] args)
10 {
11 // Variables to hold the lowest, highest, and
12 // average price of coffee.
13 double lowest = 0.0,
14 highest = 0.0,
15 average = 0.0;
```

```
16
17 // Create a named constant for the URL.
18 // NOTE: This value is specific for Java DB.
19 final String DB_URL = "jdbc:derby:CoffeeDB";
20
21 try
22 {
23 // Create a connection to the database.
24 Connection conn = DriverManager.getConnection(DB_URL);
25
26 // Create a Statement object.
27 Statement stmt = conn.createStatement();
28
29 // Create SELECT statements to get the lowest, highest,
30 // and average price from the Coffee table.
31 String minStatement = "SELECT MIN(Price) FROM Coffee";
32 String maxStatement = "SELECT MAX(Price) FROM Coffee";
33 String avgStatement = "SELECT AVG(Price) FROM Coffee";
34
35 // Get the lowest price.
36 ResultSet minResult = stmt.executeQuery(minStatement);
37 if (minResult.next())
38 lowest = minResult.getDouble(1);
39
40 // Get the highest price.
41 ResultSet maxResult = stmt.executeQuery(maxStatement);
42 if (maxResult.next())
43 highest = maxResult.getDouble(1);
44
45 // Get the average price.
46 ResultSet avgResult = stmt.executeQuery(avgStatement);
47 if (avgResult.next())
48 average = avgResult.getDouble(1);
49
50 // Display the results.
51 System.out.printf("Lowest price: %.2f\n", lowest);
52 System.out.printf("Highest price: %.2f\n", highest);
53 System.out.printf("Average price: %.2f\n", average);
54
55 // Close the connection.
56 conn.close();
57 }
58 catch(Exception ex)
59 {
60 System.out.println("ERROR: " + ex.getMessage());
61 }
62 }
63 }
```

## Program Output

```
Lowest price: $6.85
Highest price: $18.45
Average price: $10.16
```

Lines 31 through 33 declare three strings: `minStatement`, `maxStatement`, and `avgStatement`. Each of these strings contains a `SELECT` statement that uses a math function.

The code in lines 36 through 38 get the lowest price in the table. Here is a summary of how the code works:

- Line 36 executes the contents of `minStatement`, and the `ResultSet` reference that is returned is assigned to the `minResult` variable.
- The `if` statement in line 37 advances the `ResultSet` object's cursor, and line 38 gets the value of column 1 and assigns it to the `lowest` variable.

The code in lines 41 through 43 gets the highest price in the table. Here is a summary of how the code works:

- Line 41 executes the contents of `maxStatement`, and the `ResultSet` reference that is returned is assigned to the `maxResult` variable.
- The `if` statement in line 42 advances the `ResultSet` object's cursor, and line 43 gets the value of column 1 and assigns it to the `highest` variable.

The code in lines 46 through 48 gets the average price in the table. Here is a summary of how the code works:

- Line 46 executes the contents of `avgStatement`, and the `ResultSet` reference that is returned is assigned to the `avgResult` variable.
- The `if` statement in line 47 advances the `ResultSet` object's cursor, and line 48 gets the value of column 1 and assigns it to the `average` variable.



# Checkpoint

1. 15.11 What is a ResultSet object?
2. 15.12 Look at the following SQL statement.

```
SELECT Id FROM Account
```

What is the name of the table from which this statement is retrieving data?

What is the name of the column that is being retrieved?

3. 15.13 Assume a database has a table named `Inventory`, with the following columns:

| Column Name | Type     |
|-------------|----------|
| -----       |          |
| ProductID   | CHAR(10) |
| QtyOnHand   | INT      |
| Cost        | DOUBLE   |

1. Write a `SELECT` statement that will return all of the columns from every row in table.
2. Write a `SELECT` statement that will return the `ProductID` column from every row in table.
3. Write a `SELECT` statement that will return the `ProductID` column and the `QtyOnHand` column from every row in table.
4. Write a `SELECT` statement that will return the `ProductID` column only from the rows where `Cost` is less than 17.00.
5. Write a `SELECT` statement that will return all of the columns from the rows where `ProductID` ends with “ZZ”.

4. 15.14 What is the purpose of the `LIKE` operator?
5. 15.15 What is the purpose of the `%` symbol in a character pattern used by the `LIKE` operator? What is the purpose of the underline (`_`) character?
6. 15.16 How do you sort the results of a `SELECT` statement on a specific column?
7. 15.17 Assume the following declarations exist:

```
final String DB_URL = "jdbc:derby:CoffeeDB";
String sql = "SELECT * FROM Coffee";
```

Write code that uses these `String` objects to get a database connection and execute the SQL statement. Be sure to close the connection when done.

8. 15.18 How do you submit a `SELECT` statement to the DBMS?
9. 15.19 Where is a `ResultSet` object's cursor initially pointing? How do you move the cursor forward in the result set?
10. 15.20 Assume that a valid `ResultSet` object exists, populated with data. What method do you call to retrieve column 3 as a string? What do you pass as an argument to the method?

# 15.4 Inserting Rows

## Concept:

You use the `INSERT` statement in SQL to insert a new row into a table.



### VideoNote Inserting Rows

```
INSERT INTO TableName VALUES (Value1, Value2, etc...)
```

In the general format, *TableName* is the name of the database table. *Value1*, *Value2*, *etc...* is a list of column values. After the statement executes, a row containing the specified column values will be inserted into the table. Here is an example that inserts a row into the `Coffee` table, in our `CoffeeDB` database:

```
INSERT INTO Coffee VALUES ('Honduran Dark', '22-001', 8.65)
```

Notice the string values are enclosed in single-quote marks. Also, notice the order that the values appear in the list. The first value, 'Honduran Dark', is inserted into the first column of the table, which is `Description`. The second value, '22-001', is inserted into the second column of the table, which is `ProdNum`. The third value, 8.65, is inserted into the third column of the table, which is `Price`. After this statement executes, a new row will be inserted into the `Coffee` table containing the following column values:

`Description: 'Honduran Dark'`  
`ProdNum: 22-001`  
`Price: 8.95`

If you are not sure of the order in which the columns appear in the table, you

can use the following general format of the `INSERT` statement to specify the column names and their corresponding values:

```
INSERT INTO TableName
 (ColumnName1, ColumnName2, etc...)
VALUES
 (Value1, Value2, etc...)
```

In this general format, *ColumnName1*, *ColumnName2*, etc... is a list of column names, and *Value1*, *Value2*, etc... is a list of corresponding values. In the new row, *Value1* will appear in the column specified by *ColumnName1*, *Value2* will appear in the column specified by *ColumnName2*, and so forth. Here is an example:

```
INSERT INTO Coffee
 (Description, ProdNum, Price)
VALUES
 ('Honduran Dark', '22-001', 8.65)
```

This statement will produce a new row containing the following column values:

```
Description: 'Honduran Dark'
ProdNum: 22-001
Price: 8.95
```

If we rewrote the `INSERT` statement in the following manner, it would produce a new row with the same values:

```
INSERT INTO Coffee
 (ProdNum, Price, Description)
VALUES
 ('22-001', 8.65, 'Honduran Dark')
```



## Note:

If a column is a primary key, it must hold a unique value for each row in the table. No two rows in a table can have the same value in the primary key column. Recall that the `ProdNum` column is the primary key in the `Coffee`

table. The DBMS will not allow you to insert a new row with the same product number as an existing row.

# Inserting Rows with JDBC

To issue an `INSERT` statement with JDBC, you must first get a `Statement` object from the `Connection` object, using its `createStatement` method. You then use the `Statement` object's `executeUpdate` method. The method returns an `int` value representing the number of rows that were inserted into the table. Here is an example:

```
String sqlStatement = "INSERT INTO Coffee " +
 "(ProdNum, Price, Description) " +
 "VALUES ('22-001', 8.65, 'Honduran Dark')";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing an `INSERT` statement. The second statement passes this string as an argument to the `executeUpdate` method. The method should return the `int` value 1, indicating that one row was inserted into the table. The program in [Code Listing 15-7](#) shows an example. It prompts the user for the description, product number, and price of a new coffee, and inserts that data into the `Coffee` table.

## Code Listing 15-7 (`CoffeeInserter.java`)

```
1 import java.util.Scanner;
2 import java.sql.*;
3
4 /**
5 * This program lets the user insert a row into the
6 * CoffeeDB database's Coffee table.
7 */
8
9 public class CoffeeInserter
10 {
11 public static void main(String[] args)
```

```
12 {
13 String description; // To hold the coffee description
14 String prodNum; // To hold the product number
15 double price; // To hold the price
16
17 // Create a named constant for the URL.
18 // NOTE: This value is specific for Java DB.
19 final String DB_URL = "jdbc:derby:CoffeeDB";
20
21 // Create a Scanner object for keyboard input.
22 Scanner keyboard = new Scanner(System.in);
23
24 try
25 {
26 // Create a connection to the database.
27 Connection conn = DriverManager.getConnection(DB_URL);
28
29 // Get the data for the new coffee.
30 System.out.print("Enter the coffee description: ");
31 description = keyboard.nextLine();
32 System.out.print("Enter the product number: ");
33 prodNum = keyboard.nextLine();
34 System.out.print("Enter the price: ");
35 price = keyboard.nextDouble();
36
37 // Create a Statement object.
38 Statement stmt = conn.createStatement();
39
40 // Create a string with an INSERT statement.
41 String sqlStatement = "INSERT INTO Coffee " +
42 "(ProdNum, Price, Description) " +
43 "VALUES ('" +
44 prodNum + "', '" +
45 price + "', '" +
46 description + "')";
47
48 // Send the statement to the DBMS.
49 int rows = stmt.executeUpdate(sqlStatement);
50
51 // Display the results.
52 System.out.println(rows + " row(s) added to the table.");
53
54 // Close the connection.
55 conn.close();
56 }
57 catch(Exception ex)
58 {
```

```
59 System.out.println("ERROR: " + ex.getMessage());
60 }
61 }
62 }
```

## Program Output

```
Enter the coffee description: Honduran Dark
Enter the product number: 22-001
Enter the price: 8.65
1 row(s) added to the table.
```



## Checkpoint

- 15.21 Write an SQL statement to insert a new row into the Coffee table containing the following data:

Description: Eastern Blend  
ProdNum: 30-001  
Price: 18.95

- 15.22 Rewrite the following INSERT statement so it specifies the Coffee table's column names:

```
INSERT INTO Coffee
VALUES ('Honduran Dark', '22-001', 8.65)
```

# 15.5 Updating and Deleting Existing Rows

## Concept:

You use the `UPDATE` statement in SQL to change the value of an existing row. You use the `DELETE` statement to delete rows from a table.

In SQL, the `UPDATE` statement is used to change the contents of an existing row in a table. For example, if the price of Brazilian Decaf coffee changes, we could use an `UPDATE` statement to change the `Price` column for that row. Here is the general format of the `UPDATE` statement:



### VideoNote Updating and Deleting Rows

```
UPDATE Table
 SET Column = Value
 WHERE Criteria
```

In the general format, `Table` is a table name, `Column` is a column name, `Value` is a value to store in the column, and `Criteria` is a conditional expression. Here is an `UPDATE` statement that will change the price of Brazilian Decaf coffee to 9.95:

```
UPDATE Coffee
 SET Price = 9.95
 WHERE Description = 'Brazilian Decaf'
```

Here's another example:

```
UPDATE Coffee
```

```
SET Description = 'Galapagos Organic Medium'
WHERE ProdNum = '20-001'
```

This statement locates the row where `ProdNum` is '20-001' and sets the `Description` field to 'Galapagos Organic Medium'.

It's possible to update more than one row. For example, suppose we wish to change the price of every Guatemalan coffee to 12.95. If you look back at [Table 15-1](#), you will see that the product number for each of the Guatemalan coffees begins with '21'. All we need is an `UPDATE` statement that locates all the rows where the `ProdNum` column begins with '21', and changes the `Price` column of those rows to 12.95. Here is such a statement:

```
UPDATE Coffee
SET Price = 12.95
WHERE ProdNum LIKE '21%'
```



## Warning!

Be careful that you do not leave out the `WHERE` clause and the conditional expression when using an `UPDATE` statement. It's possible that you will change the contents of every row in the table! For example, look at the following statement:

```
UPDATE Coffee
SET Price = 4.95
```

Because this statement does not have a `WHERE` clause, it will change the `Price` column for every row in the `Coffee` table to 4.95!

# Updating Rows with JDBC

The process of issuing an `UPDATE` statement in JDBC is similar to that of issuing an `INSERT` statement. First, you get a `Statement` object from the `Connection` object, using its `createStatement` method. You then use the `Statement` object's `executeUpdate` method to issue the `UPDATE` statement.

The method returns an `int` value representing the number of rows affected by the `UPDATE` statement. Here is an example:

```
String sqlStatement = "UPDATE Coffee " +
 "SET Price = 9.95 " +
 "WHERE Description = 'Brazilian Decaf'";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing an `UPDATE` statement. The second statement passes this string as an argument to the `executeUpdate` method. The method returns an `int` value indicating the number of rows that were changed.

[Code Listing 15-8](#) demonstrates how to update a row in the `Coffee` table. The user enters an existing product number, and the program displays that product's data. The user then enters a new price for the specified product, and the program updates the row with the new price.

## Code Listing 15-8 (`CoffeePriceUpdater.java`)

```
1 import java.util.Scanner;
2 import java.sql.*;
3
4 /**
5 * This program lets the user change the price of a
6 * coffee in the CoffeeDB database's Coffee table.
7 */
8
9 public class CoffeePriceUpdater
10 {
11 public static void main(String[] args)
12 {
13 String prodNum; // To hold the product number
14 double price; // To hold the price
15
16 // Create a named constant for the URL.
17 // NOTE: This value is specific for Java DB.
18 final String DB_URL = "jdbc:derby:CoffeeDB";
19
20 // Create a Scanner object for keyboard input.
```

```
21 Scanner keyboard = new Scanner(System.in);
22
23 try
24 {
25 // Create a connection to the database.
26 Connection conn = DriverManager.getConnection(DB_URL);
27
28 // Create a Statement object.
29 Statement stmt = conn.createStatement();
30
31 // Get the product number for the desired coffee.
32 System.out.print("Enter the product number: ");
33 prodNum = keyboard.nextLine();
34
35 // Display the coffee's current data.
36 if (findAndDisplayProduct(stmt, prodNum))
37 {
38 // Get the new price.
39 System.out.print("Enter the new price: ");
40 price = keyboard.nextDouble();
41
42 // Update the coffee with the new price.
43 updatePrice(stmt, prodNum, price);
44 }
45 else
46 {
47 // The specified product number was not found.
48 System.out.println("That product is not found.");
49 }
50
51 // Close the connection.
52 conn.close();
53 }
54 catch(Exception ex)
55 {
56 System.out.println("ERROR: " + ex.getMessage());
57 }
58 }
59
60 /**
61 * The findAndDisplayProduct method finds a specified coffee
62 * data and displays it. The stmt parameter is a Statement
63 * object for the database. The prodNum parameter is the
64 * product number to search for. The method returns true or
65 * false indicating whether the product was found.
66 */
67
68 public static boolean findAndDisplayProduct(Statement stmt,
```

```
69 String prodNum)
70 throws SQLException
71 {
72 boolean productFound; // Flag
73
74 // Create a SELECT statement to get the specified
75 // row from the Coffee table.
76 String sqlStatement =
77 "SELECT * FROM Coffee WHERE ProdNum = '" +
78 prodNum + "'";
79
80 // Send the SELECT statement to the DBMS.
81 ResultSet result = stmt.executeQuery(sqlStatement);
82
83 // Display the contents of the result set.
84 if (result.next())
85 {
86 // Display the product.
87 System.out.println("Description: " +
88 result.getString("Description"));
89 System.out.println("Product Number: " +
90 result.getString("ProdNum"));
91 System.out.println("Price: $" +
92 result.getDouble("Price"));
93
94 // Set the flag to indicate the product was found.
95 productFound = true;
96 }
97 else
98 {
99 // Indicate the product was not found.
100 productFound = false;
101 }
102
103 return productFound;
104 }
105
106 /**
107 * The updatePrice method updates a specified coffee's price
108 * The stmt parameter is a Statement object for the database
109 * prodNum is the product number for the desired coffee.
110 * price The product's new price.
111 */
112
113 public static void updatePrice(Statement stmt, String prodN
114 double price) throws SQLException
115 {
```

```
116 // Create an UPDATE statement to update the price
117 // for the specified product number.
118 String sqlStatement = "UPDATE Coffee " +
119 "SET Price = " + Double.toString(price) +
120 "WHERE ProdNum = '" + prodNum + "'";
121
122 // Send the UPDATE statement to the DBMS.
123 int rows = stmt.executeUpdate(sqlStatement);
124
125 // Display the results.
126 System.out.println(rows + " row(s) updated.");
127 }
128 }
```

## Program Output

```
Enter the product number: 17-001
Description: Sumatra Dark
Product Number: 17-001
Price: $7.95
Enter the new price: 9.95
1 row(s) updated.
```

In the main method, line 26 gets a connection to the database, and line 29 creates a Statement object. Lines 32 and 33 prompt the user for a product number, which is assigned to the prodNum variable.

Before we let the user change the specified product's price, we want to display the product's current information. So, line 36 calls a method named findAndDisplayProduct, passing the Statement object and the prodNum variable as arguments. The findAndDisplayProduct method (which is shown in lines 68 through 104) queries the database table for the row with the specified product number. If the row is found, the method displays the row's contents then returns true. If the row is not found, the method simply returns false.

If the specified product is found, lines 39 and 40 prompt the user for the product's new price, and the user's input is assigned to the price variable. Then, line 43 calls a method named updatePrice, passing the Statement object, the prodNum variable, and the price variable as arguments. The updatePrice method (which is shown in lines 113 through 127) updates the

row containing the specified product number with the new price.

Notice neither the `findAndDisplayProduct` method nor the `updatePrice` method handle any `SQLExceptions` that might occur. If an `SQLException` happens in either of those methods, it gets passed up to the main method, where it is handled by the try-catch statement.

## Deleting Rows with the DELETE Statement

In SQL, you use the `DELETE` statement to delete one or more rows from a table. The general format of the `DELETE` statement is:

```
DELETE FROM Table WHERE Criteria
```

In the general format, `Table` is a table name and `Criteria` is a conditional expression. Here is a `DELETE` statement that will delete the row where `ProdNum` is `20-001`:

```
DELETE FROM Coffee WHERE ProdNum = '20-001'
```

This statement locates the row in the `Coffee` table where the `ProdNum` column is set to the value '`20-001`', then deletes that row.

It is possible to delete multiple rows with the `DELETE` statement. For example, look at the following statement:

```
DELETE FROM Coffee WHERE Description LIKE 'Sumatra%'
```

This statement will delete all rows in the `Coffee` table where the `Description` column begins with 'Sumatra'. If you glance back at [Table 15-1](#), you will see that four rows will be deleted.



### Warning!

Be careful that you do not leave out the WHERE clause and the conditional expression when using a DELETE statement. It's possible that you will delete every row in the table! For example, look at the following statement:

```
DELETE FROM Coffee
```

Because this statement does not have a WHERE clause, it will delete every row in the Coffee table!

## Deleting Rows with JDBC

The process of issuing a DELETE statement in JDBC is similar to that of issuing an INSERT statement or an UPDATE statement. First, you get a Statement object from the Connection object, using its createStatement method. You then use the Statement object's executeUpdate method to issue the DELETE statement. The method returns an int value representing the number of rows that were deleted. Here is an example:

```
String sqlStatement = "DELETE FROM Coffee " +
 "WHERE ProdNum = '20-001'";
int rows = stmt.executeUpdate(sqlStatement);
```

The first statement creates a string containing a DELETE statement. The second statement passes this string as an argument to the executeUpdate method. The method returns an int value indicating the number of rows that were deleted.

The program shown in [Code Listing 15-9](#) demonstrates how a row can be deleted from the Coffee table.

## Code Listing 15-9 (CoffeeDeleter.java)

```
1 import java.util.Scanner;
2 import java.sql.*;
3
```

```
4 /**
5 * This program lets the user delete a coffee
6 * from the CoffeeDB database's Coffee table.
7 */
8
9 public class CoffeeDeleter
10 {
11 public static void main(String[] args)
12 {
13 String prodNum; // To hold the product number
14 String sure; // To make sure the user wants to delete
15
16 // Create a named constant for the URL.
17 // NOTE: This value is specific for Java DB.
18 final String DB_URL = "jdbc:derby:CoffeeDB";
19
20 // Create a Scanner object for keyboard input.
21 Scanner keyboard = new Scanner(System.in);
22
23 try
24 {
25 // Create a connection to the database.
26 Connection conn = DriverManager.getConnection(DB_URL);
27
28 // Create a Statement object.
29 Statement stmt = conn.createStatement();
30
31 // Get the product number for the desired coffee.
32 System.out.print("Enter the product number: ");
33 prodNum = keyboard.nextLine();
34
35 // Display the coffee's current data.
36 if (findAndDisplayProduct(stmt, prodNum))
37 {
38 // Make sure the user wants to delete this product.
39 System.out.print("Are you sure you want to delete " +
40 "this item? (y/n): ");
41 sure = keyboard.nextLine();
42
43 if (Character.toUpperCase(sure.charAt(0)) == 'Y')
44 {
45 // Delete the specified coffee.
46 deleteCoffee(stmt, prodNum);
47 }
48 else
49 {
50 System.out.println("The item was not deleted.");
51 }
52 }
53 }
54 }
```

```
51 }
52 }
53 else
54 {
55 // The specified product number was not found.
56 System.out.println("That product is not found.");
57 }
58
59 // Close the connection.
60 conn.close();
61 }
62 catch(Exception ex)
63 {
64 System.out.println("ERROR: " + ex.getMessage());
65 }
66 }
67
68 /**
69 * The findAndDisplayProduct method finds a specified coffee
70 * data and displays it.
71 * The stmt parameter is a Statement object for the database
72 * prodNum is the product number. The method returns true or
73 * false to indicate whether the product was found.
74 */
75
76 public static boolean findAndDisplayProduct(Statement stmt,
77 String prodNum)
78 throws SQLException
79 {
80 boolean productFound; // Flag
81
82 // Create a SELECT statement to get the specified
83 // row from the Coffee table.
84 String sqlStatement =
85 "SELECT * FROM Coffee WHERE ProdNum = '" +
86 prodNum + "'";
87
88 // Send the SELECT statement to the DBMS.
89 ResultSet result = stmt.executeQuery(sqlStatement);
90
91 // Display the contents of the result set.
92 if (result.next())
93 {
94 // Display the product.
95 System.out.println("Description: " +
96 result.getString("Description"));
97 System.out.println("Product Number: " +
```

```

98 result.getString("ProdNum"));
99 System.out.println("Price: $" +
100 result.getDouble("Price"));
101
102 // Set the flag to indicate the product was found.
103 productFound = true;
104 }
105 else
106 {
107 // Indicate the product was not found.
108 productFound = false;
109 }
110
111 return productFound;
112 }
113
114 /**
115 * The deleteCoffee method deletes a specified coffee.
116 * The stmt parameter is a Statement object for the database
117 * prodNum The product number for the desired coffee.
118 */
119
120 public static void deleteCoffee(Statement stmt, String prod
121 throws SQLException
122 {
123 // Create a DELETE statement to delete the
124 // specified product number.
125 String sqlStatement = "DELETE FROM Coffee " +
126 "WHERE ProdNum = '" + prodNum + "'";
127
128 // Send the DELETE statement to the DBMS.
129 int rows = stmt.executeUpdate(sqlStatement);
130
131 // Display the results.
132 System.out.println(rows + " row(s) deleted.");
133 }
134 }
```

## Program Output

Enter the product number: **20-001**

Description: Galapagos Medium

Product Number: 20-001

Price: \$6.85

Are you sure you want to delete this item? (y/n): **y**

1 row(s) deleted.



## Checkpoint

1. 15.23 The Midnight Roastery is running a special on decaf coffee. Write an SQL statement that changes the price of all decaf coffees to 4.95.
2. 15.24 The sale on decaf coffee didn't do too well, so the Midnight Roastery has decided to stop selling decaf. Write an SQL statement that will delete all decaf coffees from the coffee table.

# 15.6 Creating and Deleting Tables

## Concept:

In SQL, the `CREATE TABLE` statement can be used to create a database table. The `DROP TABLE` statement can be used to delete a table.

The `CoffeeDB` database we have been using as our example is very simple. It has only one table, `Coffee`, which holds product information. The usefulness of this database is limited to looking up coffee descriptions, product numbers, and prices.



**VideoNote** Creating and Deleting Tables

Suppose we want to store other data in the database, such as a list of customers. To do so, we would have to add another table to the database. In SQL, you use the `CREATE TABLE` statement to create a table. Here is the general format of the `CREATE TABLE` statement:

```
CREATE TABLE TableName
 (ColumnName1 DataType1,
 ColumnName2 DataType2,
 etc...)
```

In the general format, `TableName` is the name of the table. `ColumnName1` is the name of the first column, and `DataType1` is the SQL data type for the first column. `ColumnName2` is the name of the second column, and `DataType2` is the SQL data type for the second column. This sequence repeats for all of the columns in the table. Here is an example:

```
CREATE TABLE Customer
 (Name CHAR(25),
```

```
Address CHAR(25),
City CHAR(12),
State CHAR(2),
Zip CHAR(5))
```

This statement creates a new table named `Customer`. The columns in the `Customer` table are `Name`, `Address`, `City`, `State`, and `Zip`.

You may also specify that a column is a primary key by listing the `PRIMARY KEY` qualifier after the column's data type. Recall from our earlier discussion on database organization that a primary key is a column that holds a unique value for each row and can be used to identify specific rows. When you use the `PRIMARY KEY` qualifier with a column, you should also use the `NOT NULL` qualifier. The `NOT NULL` qualifier specifies that the column must contain a value for every row. Here is an example of how we can create a `Customer` table, using the `CustomerNumber` column as the primary key:

```
CREATE TABLE Customer
(CustomerNumber CHAR(10) NOT NULL PRIMARY KEY,
 Name CHAR(25),
 Address CHAR(25),
 City CHAR(12),
 State CHAR(2),
 Zip CHAR(5))
```

This statement creates a new table named `Customer`. It has the same structure as the table created by the previous example, with one additional column, `CustomerNumber`, which is the primary key. Because `CustomerNumber` is the primary key, this column must hold a unique value for each row in the table.

## Tip:

Remember, a primary key is used to identify a specific row in a table. When selecting a column as a primary key, make sure it holds a unique value that cannot be duplicated for two rows in the table.

Take a look at the program in [Code Listing 15-10](#). When you run this program, it creates the `Customer` table in the `CoffeeDB` database, then inserts the three rows shown in [Table 15-5](#).

# Table 15-5 Rows inserted into the customer table

| CustomerNumber | Name                   | Address               | City           | State | Zip   |
|----------------|------------------------|-----------------------|----------------|-------|-------|
| 101            | Downtown Cafe          | 17 N.<br>Main Street  | Asheville      | NC    | 55515 |
| 102            | Main Street<br>Grocery | 110 E.<br>Main Street | Canton         | NC    | 55555 |
| 103            | The Coffee Place       | 101<br>Center Plaza   | Waynesville NC |       | 55516 |

## Code Listing 15-10 (CreateCustomerTable.java)

```
1 import java.sql.*; // Needed for JDBC classes
2
3 /**
4 * This program creates a Customer
5 * table in the CoffeeDB database.
6 */
7
8 public class CreateCustomerTable
9 {
10 public static void main(String[] args)
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
16 try
17 {
18 // Create a connection to the database.
19 Connection conn = DriverManager.getConnection(DB_URL);
```

```

20
21 // Get a Statement object.
22 Statement stmt = conn.createStatement();
23
24 // Make an SQL statement to create the table.
25 String sql = "CREATE TABLE Customer" +
26 "(CustomerNumber CHAR(10) NOT NULL PRIMARY KEY, " +
27 " Name CHAR(25), " +
28 " Address CHAR(25), " +
29 " City CHAR(12), " +
30 " State CHAR(2), " +
31 " Zip CHAR(5))";
32
33 // Execute the statement.
34 stmt.execute(sql);
35
36 // Add some rows to the new table.
37 sql = "INSERT INTO Customer VALUES" +
38 "('101', 'Downtown Cafe', '17 N. Main Street', " +
39 " 'Asheville', 'NC', '55515')";
40 stmt.executeUpdate(sql);
41
42 sql = "INSERT INTO Customer VALUES" +
43 "('102', 'Main Street Grocery', " +
44 " '110 E. Main Street', " +
45 " 'Canton', 'NC', '55555')";
46 stmt.executeUpdate(sql);
47
48 sql = "INSERT INTO Customer VALUES" +
49 "('103', 'The Coffee Place', '101 Center Plaza', " +
50 " 'Waynesville', 'NC', '55516')";
51 stmt.executeUpdate(sql);
52
53 // Close the connection.
54 conn.close();
55 }
56 catch (Exception ex)
57 {
58 System.out.println("ERROR: " + ex.getMessage());
59 }
60 }
61 }
```

## Removing a Table with the DROP

# TABLE Statement

Should the need arise to delete a table from a database, you can use the `DROP TABLE` statement. Here is the statement's general format:

```
DROP TABLE TableName
```

In the general format, `TableName` is the name of the table you wish to delete. For example, suppose after we created the `customer` table, we discovered that we selected the wrong data type for many of the columns. We could delete the table and then recreate it with the proper data types. The SQL statement to delete the table would be:

```
DROP TABLE Customer
```



## Checkpoint

1. 15.25 Write the SQL statement to create a table named `Book`. The `Book` table should have the columns to hold the name of the publisher, the name of the author, the number of pages, and a 10 character ISBN number.
2. 15.26 Write a statement to delete the `Book` table you created in [Checkpoint 15.24](#).

# 15.7 Creating a New Database with JDBC

## Concept:

Creating a new database with JDBC is as simple as adding an attribute to the database URL, then using SQL to create a table in the database.

In the previous section, you learned about the CREATE TABLE statement, which is used to create a new table in an existing database. But, suppose you wish to create a completely new database. With JDBC, all you must do is append the attribute ;create=true to the database URL. For example, suppose you want to create a new database named EntertainmentDB to hold data on your collection of DVDs. In Java DB, the URL you would use would be:

```
"jdbc:derby:EntertainmentDB;create=true"
```

Because we have appended the attribute ;create=true to the database URL, the program will create the database when it runs. Then, we can use a CREATE TABLE statement to create a table in the database. The program in [Code Listing 15-11](#) demonstrates how to create a database using Java DB.

## Code Listing 15-11 (BuildEntertainmentDB.java)

```
1 import java.sql.*;
2
3 /**
4 * This program shows how to create a new database
5 * using Java DB.
```

```

6 */
7
8 public class BuildEntertainmentDB
9 {
10 public static void main(String[] args)
11 throws Exception
12 {
13 final String DB_URL =
14 "jdbc:derby:EntertainmentDB;create=true";
15
16 try
17 {
18 // Create a connection to the database.
19 Connection conn =
20 DriverManager.getConnection(DB_URL);
21
22 // Create a Statement object.
23 Statement stmt = conn.createStatement();
24
25 // Create the Dvd table.
26 System.out.println("Creating the Dvd table...");
27 stmt.execute("CREATE TABLE Dvd (" +
28 "Title CHAR(25), " +
29 "Minutes INTEGER, " +
30 "Price DOUBLE)");
31
32 // Close the resources.
33 stmt.close();
34 conn.close();
35 System.out.println("Done");
36 }
37 catch(Exception ex)
38 {
39 System.out.println("ERROR: " + ex.getMessage());
40 }
41 }
42 }

```

When this program runs, the EntertainmentDB database will be created. This is because the database URL, in line 13, has the ;create=true attribute. Lines 27 through 30 then create a table named Dvd.



## Note:

When you create a new database using Java DB, you will see a folder appear on your system with the same name as the database. This folder contains the database. To delete the entire database, simply delete the folder.

# 15.8 Scrollable Result Sets

## Concept:

A scrollable result set allows random cursor movement. By default, a result set is not scrollable.

By default, `ResultSet` objects allow you to move the cursor forward only. Once the cursor has moved past a row, you cannot move the cursor backward to read that row again. If you need to move the cursor backward through the result set, you can create a *scrollable result set*. You do this when you create a `Statement` object by using an overloaded version of a `Connection` object's `createStatement` method. The method accepts two arguments. The first specifies the result set's scrolling type. You can use any of the following constants for this argument:

`ResultSet.TYPE_FORWARD_ONLY`

This is the default scrolling type. It specifies that the result set's cursor should move only forward.

`ResultSet.TYPE_SCROLL_INSENSITIVE`

This specifies that the result set should be scrollable, allowing the cursor to move forward and backward through the result set. In addition, this result set is insensitive to changes made to the database. This means that if another program or process makes changes to the database, those changes will not appear in this result set.

`ResultSet.TYPE_SCROLL_SENSITIVE`

This specifies that the result set should be scrollable, allowing the cursor to move forward and backward through the result set. In addition, this result set

is sensitive to changes made to the database. This means if another program or process makes changes to the database, those changes will appear in this result set as soon as they are made.

The second argument specifies the result set's concurrency level. You can use any of the following constants for this argument:

`ResultSet.CONCUR_READ_ONLY`

This is the default concurrency level. It specifies that result set contains a read-only version of data from the database. You cannot change the contents of the database by altering the contents of the result set.

`ResultSet.CONCUR_UPDATEABLE`

This specifies that the result set should be updateable. Changes can be made to the result set, then those changes can be saved to the database. The `ResultSet` interface specifies several methods that may be used to update the result set, then save those updates to the database. These methods allow you to make changes to the database without issuing SQL statements. For more information on these methods, see the Java API documentation.

Assuming `conn` references a `Connection` object, here is an example of the method call:

```
Statement stmt =
 conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_READ_ONLY);
```

The `Statement` object created by this code will be scrollable, insensitive to changes made to the database by other processes, and will not be updateable.

## ResultSet Navigation Methods

Once you have created a scrollable result set, you can use the following `ResultSet` methods to move the cursor:

`first()`      Moves the cursor to the first row in the result set.

|                             |                                                                                                                                                                                                                                                                                                           |
|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>last()</code>         | Moves the cursor to the last row in the result set.                                                                                                                                                                                                                                                       |
| <code>next()</code>         | Moves the cursor to the next row in the result set.                                                                                                                                                                                                                                                       |
| <code>previous()</code>     | Moves the cursor to the previous row in the result set.                                                                                                                                                                                                                                                   |
| <code>relative(rows)</code> | Moves the cursor the number of rows specified by the argument <i>rows</i> , relative to the current row. For example, the call <code>relative(2)</code> will move the cursor 2 rows forward from the current row, and <code>relative(-1)</code> will move the cursor 1 row backward from the current row. |
| <code>absolute(row)</code>  | Moves the cursor to the row specified by the integer <i>rows</i> . Remember, row numbering begins at 1, so the call <code>absolute(1)</code> will move the cursor to the first row in the result set.                                                                                                     |



## Note:

Scrollable result sets are not supported by all JDBC drivers. If your driver does not support scrollable result sets, it will throw an exception when you try to use an unsupported navigation method.

The following code shows a simple, yet practical use of some of these methods: determining the number of rows in a result set.

```
resultSet.last(); // Move to the last row
int numRows = resultSet.getRow(); // Get the current row number
resultSet.first(); // Move back to the first row
```

This code would be useful when you need to determine the number of rows in the result set before processing any of its data. The first statement moves the cursor to the last row. The second statement calls the `ResultSet` method `getRow`, which returns the row number of the current row. The third statement then moves the cursor to the first row for subsequent processing.

# 15.9 Result Set Metadata

## Concept:

Result set metadata describes the contents of a result set. The metadata can be used to determine which columns were returned when a query that is not known in advance is executed.

The term *metadata* refers to data that describes other data. A `ResultSet` object has metadata, which describes the data stored in the `ResultSet`. You can use result set metadata to determine several things about a result set, including the number of columns it contains, the names of the columns, the types of each column, and much more. Result set metadata can be very useful if you are writing an application that will submit an SQL query, and you don't know in advance what columns will be returned.

Once you have a `ResultSet` object, you can call its `getMetaData` method to get a reference to a `ResultSetMetaData` object. Assuming `resultSet` references a `ResultSet` object, here is an example:

```
ResultSetMetaData meta = resultSet.getMetaData();
```

`ResultSetMetaData` is an interface in the `java.sql` package. It specifies numerous methods, a few of which are described in [Table 15-6](#).

## Table 15-6 A few ResultSetMetaData methods

| Method | Description |
|--------|-------------|
|--------|-------------|

|                                                 |                                                                                                                                                                                         |
|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>int getColumnCount()</code>               | Returns the number of columns in the result set.                                                                                                                                        |
| <code>String getColumnName(int col)</code>      | Returns the name of the column specified by the integer <i>col</i> . The first column is column 1.                                                                                      |
| <code>String getColumnTypeNames(int col)</code> | Returns the name of the data type of the column specified by the integer <i>col</i> . The first column is column 1. The data type name returned is the database-specific SQL data type. |
| <code>int getColumnDisplaySize(int col)</code>  | Returns the display width, in characters, of the column specified by the integer <i>col</i> . The first column is column 1.                                                             |
| <code>String getTableName(int col)</code>       | Returns the name of the table associated with the column specified by the integer <i>col</i> . The first column is column 1.                                                            |

The program in [Code Listing 15-12](#) demonstrates how metadata can be used. It asks the user to enter a SELECT statement for the Coffeedb database, and then displays information about the result set as well as the result set's contents.

## Code Listing 15-12 (`MetaDataDemo.java`)

```

1 import java.sql.*;
2 import java.util.Scanner;
3
4 /**
5 * This program demonstrates result set metadata.
6 */
7
8 public class MetaDataDemo
9 {

```

```
10 public static void main(String[] args) throws Exception
11 {
12 // Create a named constant for the URL.
13 // NOTE: This value is specific for Java DB.
14 final String DB_URL = "jdbc:derby:CoffeeDB";
15
16 try
17 {
18 // Create a Scanner object for keyboard input.
19 Scanner keyboard = new Scanner(System.in);
20
21 // Get a SELECT statement from the user.
22 System.out.println("Enter a SELECT statement for " +
23 "the CoffeeDB database:");
24 String sql = keyboard.nextLine();
25
26 // Create a connection to the database.
27 Connection conn =
28 DriverManager.getConnection(DB_URL);
29
30 // Create a Statement object.
31 Statement stmt = conn.createStatement();
32
33 // Execute the query.
34 ResultSet resultSet = stmt.executeQuery(sql);
35
36 // Get the result set metadata.
37 ResultSetMetaData meta = resultSet.getMetaData();
38
39 // Display the number of columns returned.
40 System.out.println("The result set has " +
41 meta.getColumnCount() +
42 " column(s).");
43
44 // Display the column names and types.
45 for (int i = 1; i <= meta.getColumnCount(); i++)
46 {
47 System.out.println(meta.getColumnName(i) + ", " +
48 meta.getColumnTypeName(i));
49 }
50
51 // Display the contents of the rows.
52 System.out.println("\nHere are the result set rows:");
53 while (resultSet.next())
54 {
55 // Display a row.
56 for (int i = 1; i <= meta.getColumnCount(); i++)
57 {
```

```

58 System.out.print(resultSet.getString(i));
59 }
60 System.out.println();
61 }
62
63 // Close the statement and the connection.
64 stmt.close();
65 conn.close();
66 }
67 catch(Exception ex)
68 {
69 System.out.println("ERROR: " + ex.getMessage());
70 }
71 }
72 }
```

## Program Output with Example Input Shown in Bold

Enter a SELECT statement for the CoffeeDB database:

**SELECT \* FROM Coffee WHERE Price > 10.00**

The result set has 3 column(s).

DESCRIPTION, CHAR  
 PRODNUM, CHAR  
 PRICE, DOUBLE

Here are the result set rows:

|                      |        |       |
|----------------------|--------|-------|
| Sumatra Organic Dark | 17-004 | 11.95 |
| Kona Medium          | 18-001 | 18.45 |
| Kona Dark            | 18-002 | 18.45 |
| Guatemalan Decaf     | 21-002 | 10.45 |

## Program Output with Example Input Shown in Bold

Enter a SELECT statement for the CoffeeDB database:

**SELECT ProdNum FROM Coffee WHERE Price > 10.00**

The result set has 1 column(s).

PRODNUM, CHAR

Here are the result set rows:

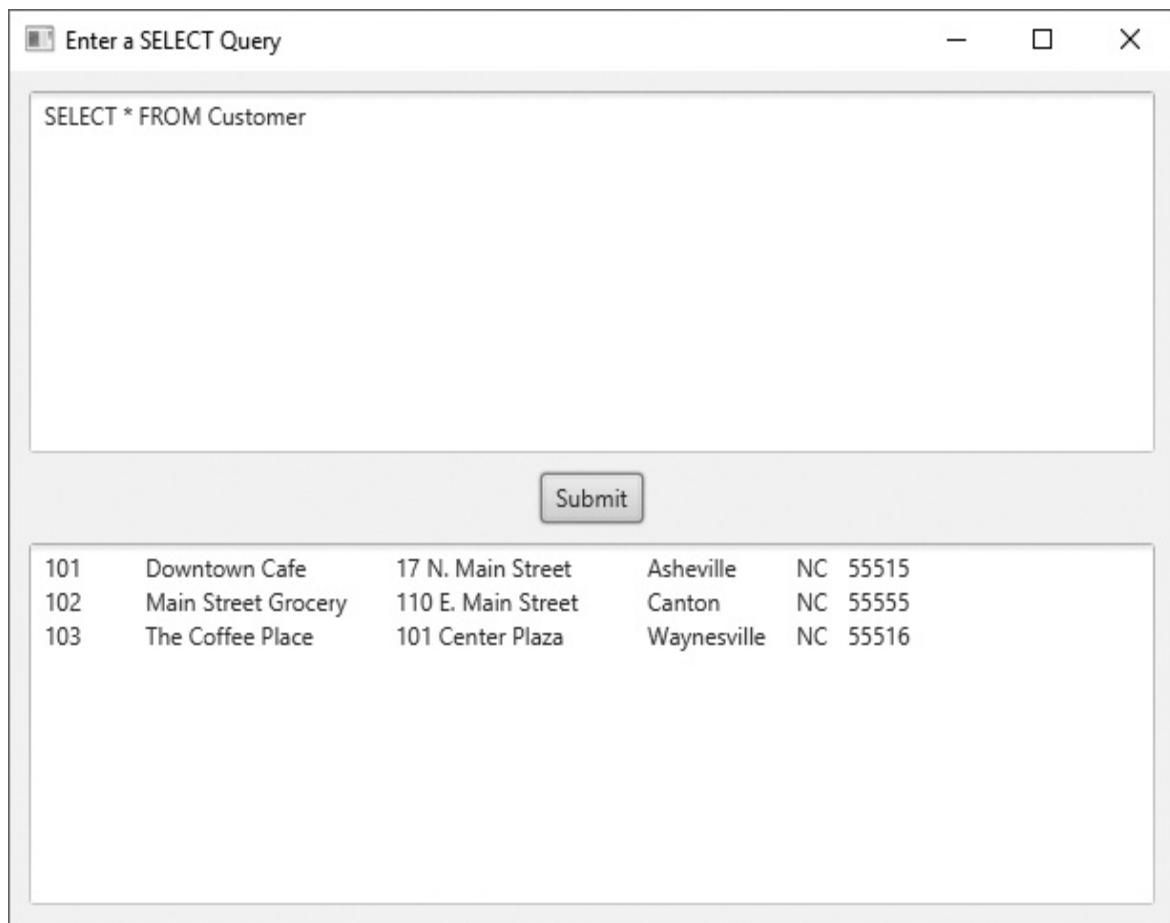
17-004  
 18-001  
 18-002  
 21-002

Line 34 submits the query to the DBMS and gets a ResultSet object. Line 37

gets a `ResultSetMetaData` object. The statement in lines 40 through 42 displays the number of columns contained in the result set. It uses the `ResultSetMetaData` object's `getColumnName` method to get this value. The loop in lines 45 through 49 iterates once for each column in the result set. Each iteration displays the column name and column data type. The `ResultSetMetaData` object's `getColumnCount` and `getColumnName` methods are used to retrieve this information. The `while` loop in lines 53 through 61 displays the contents of the result set. It has a nested `for` loop, in lines 56 through 59, which iterates once for each column in the result set. Each iteration gets the column value as a string and displays it.

Now, let's look at a JavaFX GUI application that we can use as an interactive utility to view the tables in the `CoffeeDB` database. The `DBViewer` application, which is shown in [Code Listing 15-13](#), displays the GUI shown in [Figure 15-16](#). The user enters a `SELECT` statement in the topmost `TextArea` control, then clicks the `Submit` button. The application executes the query and displays the results in the bottom `TextArea`.

## Figure 15-16 The DBViewer application



[Figure 15-16 Full Alternative Text](#)

## Code Listing 15-13 (DBViewer.java)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.TextArea;
5 import javafx.scene.control.Button;
6 import javafx.scene.layout.VBox;
7 import javafx.geometry.Insets;
8 import javafx.geometry.Pos;
9 import javafx.event.EventHandler;
10 import javafx.event.ActionEvent;
11 import java.sql.*;
12
13 public class DBViewer extends Application
14 {
```

```
15 // Database URL.
16 final String DB_URL = "jdbc:derby:CoffeeDB";
17
18 // Controls for input and output
19 TextArea queryTextArea;
20 TextArea resultsTextArea;
21
22 public static void main(String[] args)
23 {
24 // Launch the application
25 launch(args);
26 }
27
28 @Override
29 public void start(Stage primaryStage)
30 {
31 final int COL_SIZE = 50;
32 final int ROW_SIZE = 10;
33 final double SPACING = 10.0;
34
35 // Build the query entry area.
36 queryTextArea = new TextArea();
37 queryTextArea.setPrefColumnCount(COL_SIZE);
38 queryTextArea.setPrefRowCount(ROW_SIZE);
39
40 // Build the query results area.
41 resultsTextArea = new TextArea();
42 resultsTextArea.setPrefColumnCount(COL_SIZE);
43 resultsTextArea.setPrefRowCount(ROW_SIZE);
44
45 // Create the Submit button.
46 Button submitButton = new Button("Submit");
47 submitButton.setOnAction(new SubmitButtonHandler());
48
49 // Put the controls in a VBox.
50 VBox vbox = new VBox(SPACING, queryTextArea,
51 submitButton, resultsTextArea);
52 vbox.setAlignment(Pos.CENTER);
53 vbox.setPadding(new Insets(SPACING));
54
55 // Set the title.
56 primaryStage.setTitle("Enter a SELECT Query");
57
58 // Create a scene and set it to the stage.
59 Scene scene = new Scene(vbox);
60 primaryStage.setScene(scene);
61
62 // Show the stage.
```

```
63 primaryStage.show();
64 }
65 /**
66 * Event handler class for submitButton
67 */
68
69
70 class SubmitButtonHandler implements EventHandler<ActionEvent>
71 {
72 @Override
73 public void handle(ActionEvent event)
74 {
75 try
76 {
77 // Clear the previous results.
78 resultsTextArea.setText("");
79
80 // Create a connection to the database.
81 Connection conn = DriverManager.getConnection(DB_URL);
82
83 // Create a Statement object.
84 Statement stmt = conn.createStatement();
85
86 // Execute the query.
87 ResultSet resultSet =
88 stmt.executeQuery(queryTextArea.getText());
89
90 // Get the result set metadata.
91 ResultSetMetaData meta = resultSet.getMetaData();
92
93 // Create a string to hold the results
94 String output = "";
95
96 // Get the results.
97 while (resultSet.next())
98 {
99 // Get all of the columns in a row.
100 for (int i = 1; i <= meta.getColumnCount(); i++)
101 {
102 output += resultSet.getString(i);
103 output += '\t';
104 }
105 output += '\n';
106 }
107
108 // Display the results.
109 resultsTextArea.setText(output);
```

```
110 // Close the statement and the connection.
111 stmt.close();
112 conn.close();
113
114 }
115 catch (SQLException e)
116 {
117 e.printStackTrace();
118 System.exit(0);
119 }
120 }
121 }
122 }
123 }
```

# 15.10 Relational Data

## Concept:

In a relational database, a column from one table can be associated with a column from other tables. This association creates a relationship between the tables.

In Section 15.6, we added a `Customer` table to the `CoffeeDB` database. This made the database more useful by giving us the ability to look up customer information, as well as the product information held in the `Coffee` table.

Suppose we want to make the database even more useful by storing information about unpaid customer orders. That way, we can get a list of all the customers with outstanding balances. To do this, we will need to add an additional table and more data to the database. Here is a summary of the `UnpaidOrder` table, which we will create to contain order data. (We will explain what a foreign key is momentarily.)



**VideoNote** Relational Data

`UnpaidOrder` table:

|                             |                       |                    |
|-----------------------------|-----------------------|--------------------|
| <code>CustomerNumber</code> | <code>CHAR(10)</code> | <i>Foreign Key</i> |
| <code>ProdNum</code>        | <code>CHAR(10)</code> | <i>Foreign Key</i> |
| <code>OrderDate</code>      | <code>CHAR(10)</code> |                    |
| <code>Quantity</code>       | <code>DOUBLE</code>   |                    |
| <code>Cost</code>           | <code>DOUBLE</code>   |                    |

The first column, `CustomerNumber`, identifies the customer that placed the

order. Notice, however, the table does not hold any other customer data. It holds only the customer number. When designing a database, it is important that you avoid unnecessary duplication of data. Because the customer data is already stored in the Customer table, we need only to store the customer number in the Order table. We can use that number to look up the rest of the customer's data in the Customer table.

In the UnpaidOrder table, the CustomerNumber column is considered a foreign key. A *foreign key* is a column in one table that references a primary key in another table. Recall that CustomerNumber is the primary key in the Customer table. When we add a row to the UnpaidOrder table, the value that we store in the CustomerNumber column must match a value in the CustomerNumber column of the Customer table. This creates a relationship between the rows in the UnpaidOrder table and the rows in the Customer table.

The next column, ProdNum, is also a foreign key because it identifies a product in the Coffee table. This is the item the customer ordered. Once again, it is not necessary to store all of the product data in the UnpaidOrder table. We need only to store the product number, then we can use that number to look up the product data in the coffee table.

The next column in the UnpaidOrder table is OrderDate. This will hold the date that the order was placed.<sup>1</sup> The Quantity column holds the number of pounds of coffee the customer ordered. The Cost column holds the total cost of the item. We will use the following SQL statement to create the Order table:

<sup>1</sup>In SQL, there is a DATE data type which is used to hold dates. It corresponds to the java.sql.Date class. To keep the example simple, however, we will merely store the invoice date as a string.

```
CREATE TABLE UnpaidOrder
(CustomerNumber CHAR(10) NOT NULL REFERENCES Customer(CustomerNu
 ProdNum CHAR(10) NOT NULL REFERENCES Coffee(ProdNum),
 OrderDate CHAR(10),
 Quantity DOUBLE,
 Cost DOUBLE)
```

Notice this statement introduces a new qualifier, REFERENCES, which is used with both the CustomerNumber and ProdNum columns. Here is the way it is used with the CustomerNumber column:

```
REFERENCES Customer(CustomerNumber)
```

This indicates that the column references the CustomerNumber column in the Customer table. Because of this qualifier, the DBMS performs a check when you insert a row into the UnpaidOrder table. It will only allow you to insert a row if the CustomerNumber column contains a valid value from the CustomerNumber column of the Customer table. This ensures *referential integrity* between the two tables.

The REFERENCES qualifier is also used with the ProdNum column:

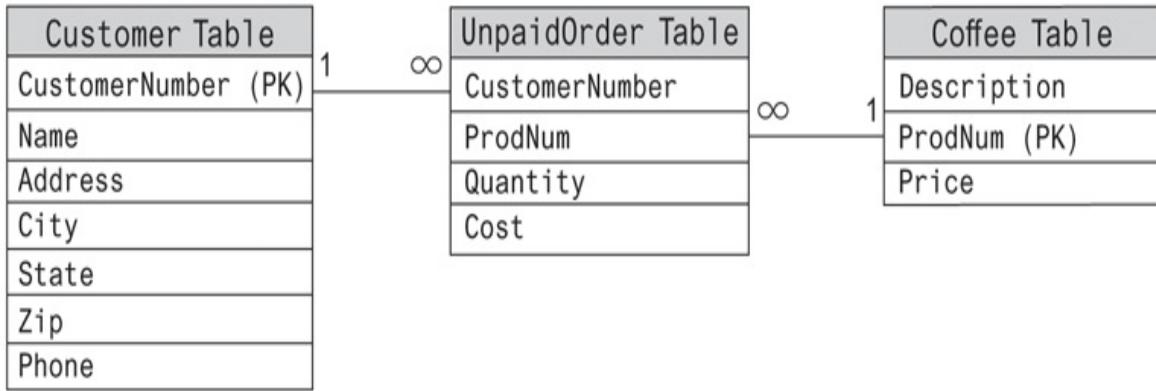
```
REFERENCES Coffee(ProdNum)
```

This indicates that the column references the ProdNum column in the Coffee table. When you insert a row into the Order table, its ProdNum column must contain a valid value from the ProdNum column of the Coffee table.

System designers commonly use *entity relationship diagrams* to show the relationships between database tables. [Figure 15-17](#) shows an entity relationship diagram for the coffeeDB database. In the diagram, the primary keys are denoted with (PK). The lines that are drawn between the tables show how the tables are related. In this diagram, there are two types of relationships:

- A *one to many relationship* means for each row in table A, there can be many rows in table B that reference it.
- A *many to one relationship* means many rows in table A can reference a single row in table B.

## Figure 15-17 Entity relationship diagram



[Figure 15-17 Full Alternative Text](#)

Notice the ends of each line show either a 1 or an infinity symbol ( $\infty$ ). You can interpret the infinity symbol as meaning *many*, and the number 1 as meaning *one*. Look at the line that connects the Customer table to the UnpaidOrder table. The 1 is at the end of the line near the Customer table, and the infinity symbol is at the end near the UnpaidOrder table. This means that one row in the Customer table may be referenced by many rows in the UnpaidOrder table. This makes sense, because a customer can place many orders. (In fact, this is what management hopes for!)

If we look at the relationship in the other direction, we see that many of the rows in the UnpaidOrder table can reference one row in the Customer table. Here is a summary of all the relationships shown in the diagram:

- There is a one to many relationship between the Customer table and the UnpaidOrder table. One row in the Customer table may be referenced by many rows in the UnpaidOrder table.
- There is a many to one relationship between the UnpaidOrder table and the Customer table. Many rows in the UnpaidOrder table may reference a single row in the Customer table.
- There is a one to many relationship between the Coffee table and the UnpaidOrder table. One row in the Coffee table may be referenced by many rows in the UnpaidOrder table.
- There is a many to one relationship between the UnpaidOrder table and

the `Coffee` table. Many rows in the `UnpaidOrder` table may reference a single row in the `Coffee` table.

# Joining Data from Multiple Tables

When related data is stored in multiple tables, as in the `CoffeeDB` database, it is often necessary to pull data from different tables in a `SELECT` statement. For example, suppose we want to see information about all the unpaid orders. Specifically, for each unpaid order, we want to see the customer number, customer name, order date, coffee description, and cost. This involves columns from the `Customer`, `UnpaidOrder`, and `Coffee` tables. Because some of these tables have columns with the same name, we have to use qualified column names in our `SELECT` statement. A *qualified column name* takes the following form:

`TableName.ColumnName`

For example, `Customer.CustomerNumber` specifies the `CustomerNumber` column in the `Customer` table, and `UnpaidOrder.CustomerNumber` specifies the `CustomerNumber` column in the `UnpaidOrder` table. Take a look at the following query:

```
SELECT
 Customer.CustomerNumber,
 Customer.Name,
 UnpaidOrder.OrderDate,
 Coffee.Description,
 UnpaidOrder.Cost
FROM
 Customer, UnpaidOrder, Coffee
WHERE
 UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
 UnpaidOrder.ProdNum = Coffee.ProdNum
```

The first part of the query specifies the columns we want:

```
SELECT
 Customer.CustomerNumber,
 Customer.Name,
 UnpaidOrder.OrderDate,
```

```
Coffee.Description,
UnpaidOrder.Cost
```

The second part of the query, which uses the `FROM` clause, specifies the tables we want to pull the data from:

```
FROM
Customer, UnpaidOrder, Coffee
```

Notice the table names are separated by commas. The third part of the query, which uses the `WHERE` clause, specifies a search criteria:

```
WHERE
UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
UnpaidOrder.ProdNum = Coffee.ProdNum
```

The search criteria tell the DBMS how to link the rows in the tables. Recall from our earlier discussion that `UnpaidOrder.CustomerNumber` column references the `Customer.CustomerNumber` column, and the `UnpaidOrder.ProdNum` column references the `Coffee.ProdNum` column.



## Warning!

When joining data from multiple tables, be sure to use a `WHERE` clause to specify a search criteria that links the appropriate columns. Failure to do so will result in a large set of unrelated data.

# An Order Entry System

Now, let's look at an example application that uses a relational database. In order to use this application, you will need the `Coffee` table, the `Customer` table, and the `UnpaidOrder` table in the `CoffeeDB` database. Back in Section 15.6, you saw a program named `CreateCustomerTable.java`. This program created the `Customer` table in the `CoffeeDB` database and added three sample rows. If you haven't already run that program, do so now. After running the program, you can run the `DBViewer` application presented earlier in this

chapter, and enter the statement `SELECT * FROM Customer`. You should see the data shown in [Figure 15-18](#).

## Figure 15-18 Customer table displayed in the DBViewer application

The screenshot shows the DBViewer application window. The title bar says "Enter a SELECT Query". The main area contains a SQL query:

```
SELECT
 Customer.CustomerNumber,
 Customer.Name,
 UnpaidOrder.OrderDate,
 Coffee.Description,
 UnpaidOrder.Cost
FROM
 Customer, UnpaidOrder, Coffee
WHERE
 UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
 UnpaidOrder.ProdNum = Coffee.ProdNum
```

Below the query is a "Submit" button. A result table is displayed with the following data:

|  | 102 | Main Street Grocery | 12/15/2018 | Brazilian Medium | 79.5 |
|--|-----|---------------------|------------|------------------|------|
|  |     |                     |            |                  |      |

Next, you should run the program `CreateUnpaidOrderTable.java`. This program is in the source code folder for this chapter, and it will create the `UnpaidOrder` table we discussed earlier in this chapter. The table will have no data stored in it, however.

Now that we have the necessary tables set up in our database, we will

examine an order entry application that allows the user to place an order for coffee. The application is built from two classes: one to manage the database, and another to display the GUI. The first class we will look at is the CoffeeDBManager class, shown in [Code Listing 15-14](#). This class provides several static methods that perform a variety of operations on the CoffeeDB database, which we will need in our order entry system.

## Code Listing 15-14 (CoffeeDBManager.java)

```
1 import java.util.ArrayList;
2 import java.sql.*;
3
4 /**
5 * The CoffeeDBManager class performs operations
6 * the CoffeeDB database.
7 */
8
9 public class CoffeeDBManager
10 {
11 // Constant for database URL.
12 public final static String DB_URL = "jdbc:derby:CoffeeDB";
13
14 /**
15 * The getCustomerNames method returns an ArrayList
16 * of Strings containing all the customer names.
17 */
18 public static ArrayList<String>getCustomerNames() throws SQ
19 {
20 // Create a connection to the database.
21 Connection conn = DriverManager.getConnection(DB_URL);
22
23 // Create a connection to the database.
24 conn = DriverManager.getConnection(DB_URL);
25
26 // Create a Statement object for the query.
27 Statement stmt =
28 conn.createStatement(
29 ResultSet.TYPE_SCROLL_SENSITIVE,
30 ResultSet.CONCUR_READ_ONLY);
31
32 // Execute the query.
```

```
33 ResultSet resultSet = stmt.executeQuery(
34 "SELECT Name FROM Customer");
35
36 // Get the number of rows
37 resultSet.last(); // Move last row
38 int numRows = resultSet.getRow(); // Get row number
39 resultSet.first(); // Move to first row
40
41 // Create an ArrayList for the customer names.
42 ArrayList<String>listData = new ArrayList<>();
43
44 // Populate the ArrayList with customer names.
45 for (int index = 0; index <numRows; index++)
46 {
47 // Store the customer name in the array.
48 listData.add(resultSet.getString(1));
49
50 // Go to the next row in the result set.
51 resultSet.next();
52 }
53
54 // Close the connection and statement objects.
55 conn.close();
56 stmt.close();
57
58 // Return the listData array.
59 return listData;
60 }
61
62 /**
63 * The getCustomerNames method returns an array
64 * of Strings containing all the customer names.
65 */
66 public static ArrayList<String>getCustomerNames() throws SQLException
67 {
68 // Create a connection to the database.
69 Connection conn = DriverManager.getConnection(DB_URL);
70
71 // Create a Statement object for the query.
72 Statement stmt =
73 conn.createStatement(
74 ResultSet.TYPE_SCROLL_SENSITIVE,
75 ResultSet.CONCUR_READ_ONLY);
76
77 // Execute the query.
78 ResultSet resultSet = stmt.executeQuery(
79 "SELECT Description FROM Coffee");
80 }
```

```
81 // Get the number of rows
82 resultSet.last(); // Move to last row
83 int numRows = resultSet.getRow(); // Get row number
84 resultSet.first(); // Move to first row
85
86 // Create an array for the coffee names.
87 ArrayList<String>listData = new ArrayList<>();
88
89 // Populate the array with coffee names.
90 for (int index = 0; index <numRows; index++)
91 {
92 // Store the coffee name in the array.
93 listData.add(resultSet.getString(1));
94
95 // Go to the next row in the result set.
96 resultSet.next();
97 }
98
99 // Close the connection and statement objects.
100 conn.close();
101 stmt.close();
102
103 // Return the listData array.
104 return listData;
105 }
106
107 /**
108 * The getProdNum method returns a specific
109 * coffee's product number.
110 */
111 public static String getProdNum(String coffeeName) throws S
112 {
113 String prodNum = ""; // Product number
114
115 // Create a connection to the database.
116 Connection conn = DriverManager.getConnection(DB_URL);
117
118 // Create a Statement object for the query.
119 Statement stmt = conn.createStatement();
120
121 // Execute the query.
122 ResultSet resultSet = stmt.executeQuery(
123 "SELECT ProdNum " +
124 "FROM Coffee " +
125 "WHERE Description = '" +
126 coffeeName + "'");
```

```
128 // If the result set has a row, go to it
129 // and retrieve the product number.
130 if (resultSet.next())
131 prodNum = resultSet.getString(1);
132
133 // Close the Connection and Statement objects.
134 conn.close();
135 stmt.close();
136
137 // Return the product number.
138 return prodNum;
139 }
140
141 /**
142 * The getCoffeePrice method returns the price
143 * of a coffee.
144 */
145 public static double getCoffeePrice(String prodNum) throws
146 {
147 double price = 0.0; // Coffee price
148
149 // Create a connection to the database.
150 Connection conn = DriverManager.getConnection(DB_URL);
151
152 // Create a Statement object for the query.
153 Statement stmt = conn.createStatement();
154
155 // Execute the query.
156 ResultSet resultSet = stmt.executeQuery(
157 "SELECT Price " +
158 "FROM Coffee " +
159 "WHERE ProdNum = '" +
160 prodNum + "'");
161
162 // If the result set has a row, go to it
163 // and retrieve the price.
164 if (resultSet.next())
165 price = resultSet.getDouble(1);
166
167 // Close the connection and statement objects.
168 conn.close();
169 stmt.close();
170
171 // Return the price.
172 return price;
173 }
174
175 /**
```

```
176 * The getCustomerNum method returns a specific
177 * customer's number.
178 */
179 public static String getCustomerNum(String name) throws SQL
180 {
181 String customerNumber = " ";
182
183 // Create a connection to the database.
184 Connection conn = DriverManager.getConnection(DB_URL);
185
186 // Create a Statement object for the query.
187 Statement stmt = conn.createStatement();
188
189 // Execute the query.
190 ResultSet resultSet = stmt.executeQuery(
191 "SELECT CustomerNumber " +
192 "FROM Customer " +
193 "WHERE Name = '" + name + "'");
194
195 if (resultSet.next())
196 customerNumber = resultSet.getString(1);
197
198 // Close the connection and statement objects.
199 conn.close();
200 stmt.close();
201
202 // Return the customer number.
203 return customerNumber;
204 }
205
206 /**
207 * The submitOrder method submits an order to
208 * the UnpaidOrder table in the CoffeeDB database.
209 */
210 public static void submitOrder(String custNum, String prodN
211 int quantity, double price,
212 String orderDate) throws SQL
213 {
214 // Calculate the cost of the order.
215 double cost = quantity * price;
216
217 // Create a connection to the database.
218 Connection conn = DriverManager.getConnection(DB_URL);
219
220 // Create a Statement object for the query.
221 Statement stmt = conn.createStatement();
222 }
```

```

223 // Execute the query.
224 stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('"
225 + custNum + "', '" +
226 + prodNum + "', '" + orderDate + "', " +
227 + quantity + ", " + cost + ")");
228
229 // Close the connection and statement objects.
230 conn.close();
231 stmt.close();
232 }
233 }
```

Here is a summary of the methods in the `CoffeeDBManager` class:

- The `getCustomerNames` method, in lines 18 through 60, returns an `ArrayList` of strings containing the names of all the customers in the `Customer` table.
- The `getCoffeeNames` method, in lines 66 through 105, returns an `ArrayList` of strings containing the names of all the coffees in the `Coffee` table.
- The `getProdNum` method, in lines 111 through 139, accepts a `String` argument containing the name of a coffee. The method returns the coffee's product number.
- The `getCoffeePrice` method, in lines 145 through 173, accepts a `String` argument containing a coffee's product number. The method returns the price of the specified coffee.
- The `getCustomerNum` method, in lines 179 through 204, accepts a `String` argument containing a customer's name. The method returns that customer's customer number.
- The `submitOrder` method, in lines 210 through 232, creates a row in the `UnpaidOrder` table. It accepts arguments for the customer number, the product number of the coffee being ordered, the quantity being ordered, the coffee's price per pound, and the order date. Line 215 calculates the cost of the order by multiplying the quantity by the price per pound. Line 218 opens a connection to the database, and line 221 creates a

Statement object. Lines 224 through 227 execute an `INSERT` statement on the `UnpaidOrders` table.

Next, we will look at the `OrderEntrySystem` class, shown in [Code Listing 15-15](#). This class builds the GUI and interacts with the user. When you run the application, it displays the GUI shown in [Figure 15-17](#). To place an order, you select a customer and a coffee from the `ListView` controls, enter the quantity and the date in the `TextField` controls, and click the `Submit` button. The system then inserts a new row for the order into the `UnpaidOrder` table.

## Code Listing 15-15 (`OrderEntrySystem.java`)

```
1 import javafx.application.Application;
2 import javafx.stage.Stage;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Label;
5 import javafx.scene.control.ListView;
6 import javafx.scene.control.TextField;
7 import javafx.scene.control.Button;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.layout.HBox;
10 import javafx.scene.layout.BorderPane;
11 import javafx.geometry.Insets;
12 import javafx.geometry.Pos;
13 import java.sql.SQLException;
14 import javafx.event.EventHandler;
15 import javafx.event.ActionEvent;
16
17 public class OrderEntrySystem extends Application
18 {
19 // Fields for input
20 private ListView<String> customerListView;
21 private ListView<String> coffeeListView;
22 private TextField qtyTextField;
23 private TextField orderDateTextField;
24
25 // Constants
26 final int LV_WIDTH = 150;
27 final int LV_HEIGHT = 75;
28 final double SPACING = 10.0;
```

```
29
30 private void buildGUI(Stage stage)
31 {
32 // Build the customer selection area.
33 Label customerPrompt = new Label("Select a Customer");
34 customerListView = new ListView<String>();
35 customerListView.setPrefSize(LV_WIDTH, LV_HEIGHT);
36 VBox customerVBox = new VBox(SPACING, customerPrompt,
37 customerListView);
38 customerVBox.setAlignment(Pos.CENTER);
39 customerVBox.setPadding(new Insets(SPACING));
40
41 // Build the coffee selection area
42 Label coffeePrompt = new Label("Select a Coffee");
43 coffeeListView = new ListView<String>();
44 coffeeListView.setPrefSize(LV_WIDTH, LV_HEIGHT);
45 VBox coffeeVBox = new VBox(10, coffeePrompt,
46 coffeeListView);
47 coffeeVBox.setAlignment(Pos.CENTER);
48 coffeeVBox.setPadding(new Insets(SPACING));
49
50 // Build the quantity and date entry areas.
51 Label qtyPrompt = new Label("Quantity");
52 qtyTextField = new TextField();
53
54 Label datePrompt = new Label("Order Date");
55 orderDateTextField = new TextField();
56 VBox orderVBox = new VBox(SPACING, qtyPrompt, qtyTextField
57 datePrompt, orderDateTextField);
58 orderVBox.setAlignment(Pos.CENTER);
59 orderVBox.setPadding(new Insets(SPACING));
60
61 // Build the button area.
62 Button submitButton = new Button("Submit");
63 Button exitButton = new Button("Exit");
64 HBox buttonHBox = new HBox(SPACING, submitButton, exitButton);
65 buttonHBox.setAlignment(Pos.CENTER);
66 buttonHBox.setPadding(new Insets(SPACING));
67
68 // Register event handlers for the Button controls.
69 submitButton.setOnAction(new SubmitButtonHandler());
70 exitButton.setOnAction(e ->
71 {
72 // Close the stage, end the application.
73 stage.close();
74 });
75
76 // Put everything inside a BorderPane.
```

```
77 BorderPane borderPane = new BorderPane();
78 borderPane.setLeft(customerVBox);
79 borderPane.setCenter(coffeeVBox);
80 borderPane.setRight(orderVBox);
81 borderPane.setBottom(buttonHBox);
82
83 // Set the title bar text.
84 stage.setTitle("Order Entry System");
85
86 // Create a scene and add it to the stage
87 Scene scene = new Scene(borderPane);
88 stage.setScene(scene);
89 }
90
91 /**
92 * The loadData method loads customer names and
93 * coffee names into the ListView controls.
94 */
95
96 private void loadData()
97 {
98 try
99 {
100 // Load customer names into the ListView.
101 customerListView.getItems().setAll(
102 CoffeeDBManager.getCustomerNames());
103
104 // Load coffee names into the ListView.
105 coffeeListView.getItems().setAll(
106 CoffeeDBManager.getCoffeeNames());
107 }
108 catch (SQLException e)
109 {
110 e.printStackTrace();
111 System.exit(0);
112 }
113 }
114
115 public static void main(String[] args)
116 {
117 // Launch the application
118 launch(args);
119 }
120
121 @Override
122 public void start(Stage primaryStage)
123 {
```



```

172
173 // Clear the text fields
174 qtyTextField.clear();
175 orderDateTextField.clear();
176 }
177 catch (SQLException e)
178 {
179 e.printStackTrace();
180 System.exit(0);
181 }
182 }
183 }
184 }
```

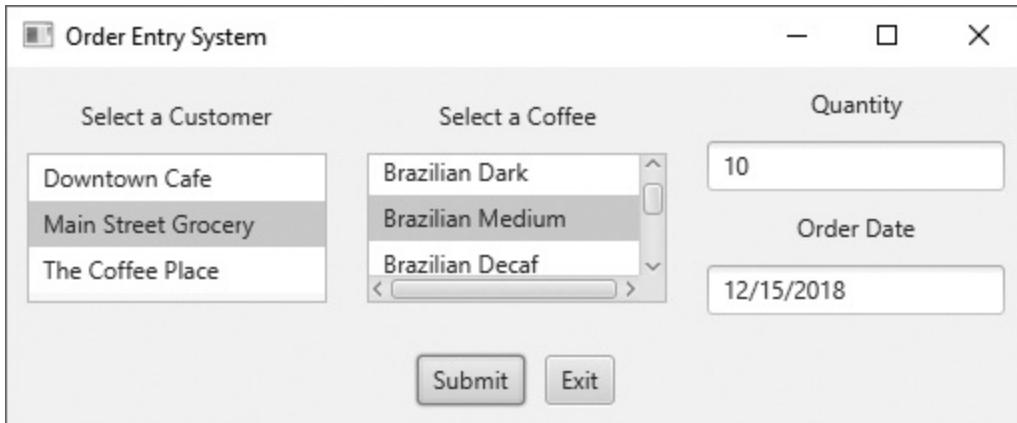
Let's take a closer look at the code.

- Lines 20 through 23 declare the `ListView` and `TextField` fields that will be used for input.
- Lines 26 through 28 declare constants for the `ListView` width and height, and for the spacing that we will place around controls in our layout containers.
- The `buildGUI` method in lines 30 through 89 creates all of the controls, places them in layout containers, and registers event handlers for the buttons. A `Scene` is created, and it is set to the stage.
- The `loadData` method, in lines 96 through 113, uses the `CoffeeDBManager` class to get the customer names and coffee names from the database and load them into the `ListView` controls.
- The `start` method, in lines 122 through 132, calls the `buildGUI` method (line 125), calls the `loadData` method (line 128), and shows the stage (line 131).
- The `SubmitButtonHandler` class, in lines 138 through 183, is the event handler for the `submitButton` control. It's `handle` method performs the following steps:
  - It gets the selected customer name from the `customerListView` control in lines 146 and 147.

- It gets the selected coffee name from the `coffeeListView` control in lines 150 and 151.
- It gets the quantity from the `qtyTextField` control in line 154.
- It gets the order date from the `orderDateTextField` in line 157.
- At this point, we have the name of the customer placing the order, and the name of the coffee being ordered, but to submit an order we need the customer number and the product number. Lines 160 and 161 call the `CoffeeDBManager` class's `getCustomerNum` method to retrieve the customer number from the database. Line 164 calls the `CoffeeDBManager` class's `getProdNum` method to retrieve the product number. We also need the price of the coffee. Line 167 calls the `CoffeeDBManager` class's `getCoffeePrice` method to retrieve this information.
- Lines 170 and 171 call the `CoffeeDBManager` class's `submitOrder` method to submit the order. After the order is submitted, lines 174 and 175 clear the text fields holding the quantity and order date, making it easier to enter the next order.

[Figure 15-19](#) shows the order entry GUI with a customer selected, a coffee selected, a quantity entered, and an order date entered.

## Figure 15-19 Order data entered



After we submit the order shown in [Figure 15-19](#), we can run the DBViewer application and enter the following SELECT statement to pull data from various tables relating to the order. [Figure 15-20](#) shows the DBViewer application's screen with the SELECT statement already filled in and the results of the statement.

## Figure 15-20 Order information viewed in DBViewer

Enter a SELECT Query

```
SELECT
 Customer.CustomerNumber,
 Customer.Name,
 UnpaidOrder.OrderDate,
 Coffee.Description,
 UnpaidOrder.Cost
FROM
 Customer, UnpaidOrder, Coffee
WHERE
 UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
 UnpaidOrder.ProdNum = Coffee.ProdNum
```

Submit

|     |                     |            |                  |      |
|-----|---------------------|------------|------------------|------|
| 102 | Main Street Grocery | 12/15/2018 | Brazilian Medium | 79.5 |
|-----|---------------------|------------|------------------|------|

```
SELECT
 Customer.CustomerNumber,
 Customer.Name,
 UnpaidOrder.OrderDate,
 Coffee.Description,
 UnpaidOrder.Cost
FROM
 Customer, UnpaidOrder, Coffee
WHERE
 UnpaidOrder.CustomerNumber = Customer.CustomerNumber AND
 UnpaidOrder.ProdNum = Coffee.ProdNum
```

## 15.11 Advanced Topics

### Transactions

Sometimes an application must perform several database updates to carry out a single task. For example, suppose you have a checking account and a car loan at your bank. Each month, your car payments are automatically taken from your checking account. For this operation to take place, the balance of your checking account must be reduced by the amount of the car payment, and the balance of the car loan must also be reduced.

An operation that requires multiple database updates is known as a *transaction*. In order for a transaction to be complete, all of the steps involved in the transaction must be performed. If any single step within a transaction fails, then none of the steps in the transaction should be performed. For example, imagine that the bank system has begun the process of making your car payment. The amount of the payment is subtracted from your checking account balance, but then some sort of system failure occurs before the balance of the car loan is reduced. You would be quite upset to learn that the amount for your car payment was withdrawn from your checking account, but never applied to your loan!

Most database systems provide a means for undoing the partially completed steps in a transaction when a failure occurs. When you write transaction-processing code, there are two concepts you must understand: commit and rollback. The term *commit* refers to making a permanent change to a database, and the term *rollback* refers to undoing changes to a database.

By default, the JDBC connection class operates in auto-commit mode. In *auto-commit* mode, all updates that are made to the database are made permanent as soon as they are executed. When auto-commit mode is turned off, however, changes do not become permanent until a commit command is executed. This makes it possible to use a rollback command to undo changes. A rollback command will undo all database changes since the last commit.

command.

In JDBC, you turn auto-commit mode off with the `Connection` class's `setAutoCommit` method, passing the argument `false`. Here is an example:

```
conn.setAutoCommit(false);
```

You execute a commit command by calling the `Connection` class's `commit` method, as shown here:

```
conn.commit();
```

A rollback command can be executed with the `Connection` class's `rollback` method, as shown here:

```
conn.rollback();
```

Let's look at an example. Suppose we add a new table named `Inventory` to the `CoffeeDB` database, for the purpose of storing the quantity of each type of coffee in inventory. The table has two rows: `ProdNum`, which is a coffee product number, and `Qty`, which is the quantity of each type of coffee. When an order is placed, we want to update both the `Inventory` table and the `UnpaidOrder` table. In the `Inventory` table, we want to subtract the quantity being ordered from the quantity in inventory. In the `UnpaidOrder` table, we want to insert a new row representing the order. Here is some example code that might be used to process this as a transaction.

```
Connection conn = DriverManager.getConnection(DB_URL);
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
// Attempt the transaction.
try
{
 // Update the inventory records.
 stmt.executeUpdate("UPDATE Inventory SET Qty = Qty - " +
 qtyOrdered + " WHERE ProdNum = " + prodNum);
 // Add the order to the UnpaidOrder table.
 stmt.executeUpdate("INSERT INTO UnpaidOrder VALUES('" +
 custNum + "', '" +
 prodNum + "', '" + orderDate + "', " +
 qtyOrdered + ", " + cost + "')");
```

```
// Commit all these updates.
conn.commit();
}
catch (SQLException ex)
{
 // Roll back the changes.
 conn.rollback();
}
```

Notice that inside the try block, after the statements to update the database have been executed, the `Connection` class's `commit` method is executed. In the catch block, the `rollback` method is executed in the event of an error.

## Stored Procedures

Many commercial database systems allow you to create SQL statements and store them in the DBMS itself. These SQL statements are called *stored procedures*, and they can be executed by other applications using the DBMS. If you have written an SQL statement that is used often in a variety of applications, it might be helpful to store it as a stored procedure in the DBMS. Then, you can call the stored procedure from any of your applications when you need to execute the SQL statement. Because stored procedures are already in the DBMS, they usually execute faster than SQL statements that are submitted from applications outside the DBMS.

We won't go into the details of stored procedures in this book, but we will point you in the right direction if you want to learn more. Each DBMS has its own syntax for creating a stored procedure in SQL, so you will have to consult your DBMS documentation to determine the format. Once you have properly written a stored procedure in SQL, you simply submit it to the DBMS using the `Statement` class's `execute` method. To execute a stored procedure, you must create a `CallableStatement` object. `CallableStatement` is an interface in the `java.sql` package. To create a `CallableStatement` object, you call the `Connection` class's `prepareCall` statement.

## 15.12 Common Errors to Avoid

- Using the == operator instead of the = **operator in an SQL statement.** The equal-to operator in SQL is one = sign, instead of two.
- Using double quotes around strings instead of single quotes. String literals in SQL are enclosed in single quotes instead of double quotes.
- Using && and || in an SQL statement. The logical AND and logical OR operators in SQL are the words AND and OR, not the && and || symbols.
- Not using the correct WHERE clause in an UPDATE statement. Be careful that you do not leave out the WHERE clause and the conditional expression when using an UPDATE statement. It's possible that you will change the contents of every row in the table!
- Not using the correct WHERE clause in a DELETE statement. Be careful that you do not leave out the WHERE clause and the conditional expression when using a DELETE statement. It's possible that you will delete every row in the table!
- Not using the correct WHERE clause when joining data. When joining data from multiple tables, be sure to use a WHERE clause to specify a search criteria that links the appropriate columns. Failure to do so will result in a large set of unrelated data.

# **Review Questions and Exercises**

## **Multiple Choice and True/False**

1. This is the technology that makes it possible for a Java application to communicate with a DBMS.
  1. DBMSC
  2. JDBC
  3. JDBMS
  4. JDSQL
2. This is a standard language for working with database management systems.
  1. Java
  2. COBOL
  3. SQL
  4. BASIC
3. The data that is stored in a table is organized in .
  1. rows
  2. files
  3. folders
  4. pages

4. The data that is stored in a row is divided into .
1. sections
  2. bytes
  3. columns
  4. tables
5. This is a column that holds a unique value for each row and can be used to identify specific rows.
1. ID column
  2. public key
  3. designator column
  4. primary key
6. This type of SQL statement is used to retrieve rows from a table.
1. RETRIEVE
  2. GET
  3. SELECT
  4. READ
7. This contains the results of an SQL SELECT statement.
1. select set
  2. result set
  3. SQL set
  4. collection set

8. This clause allows you to specify a search criteria with the SELECT statement.
  1. SEARCH
  2. WHERE
  3. AS
  4. CRITERIA
9. This is a Java class that is designed to communicate with a specific DBMS.
  1. JDBC driver
  2. DBMS Superclass
  3. DBMS Subclass
  4. Stream converter
10. This is a string listing the protocol that should be used to access a database, the name of the database, and potentially other items.
  1. JDBC driver
  2. JDBC locator
  3. Database URL
  4. Database specifier
11. This method is specified in the Statement interface and should be used to execute a SELECT statement.
  1. execute
  2. executeUpdate

- 3. executeQuery
  - 4. executeSelect
- 12. This method is specified in the Statement interface and should be used to execute an UPDATE statement.
  - 1. execute
  - 2. executeUpdate
  - 3. executeQuery
  - 4. executeSelect
- 13. This method is specified in the Statement interface and should be used to execute an INSERT statement.
  - 1. execute
  - 2. executeUpdate
  - 3. executeQuery
  - 4. executeSelect
- 14. This SQL statement is used to insert rows into a table.
  - 1. INSERT
  - 2. ADD
  - 3. CREATE
  - 4. UPDATE
- 15. This SQL statement is used to remove rows from a table.
  - 1. REMOVE

2. ERASE
  3. PURGE
  4. DELETE
16. This SQL statement is used to delete an entire table.
1. REMOVE
  2. DROP
  3. PURGE
  4. DELETE
17. This is a column in one table that references a primary key in another table.
1. secondary key
  2. fake key
  3. foreign key
  4. duplicate key
18. True or False: Java comes with its own DBMS.
19. True or False: A Java application that uses a DBMS to store data does not need know about the physical structure of the data.
20. True or False: You use SQL instead of Java to write entire applications, including the user interface.
21. True or False: In SQL, the not-equal-to operator is !=, which is the same as in Java.
22. True or False: When a ResultSet object is initially created, its cursor is

pointing at the first row in the result set.

23. True or False: In a transaction, it is permissible for only some of the database updates to be made.
24. True or False: The term *rollback* refers to undoing changes to a database.

## Find the Error

1. Find the error in the following SQL statement.

```
SELECT * FROM Coffee WHERE Description = "French Roast Dark"
```

2. Find the error in the following SQL statement.

```
SELECT * FROM Coffee WHERE ProdNum != '14-001'
```

3. Find the error in the following Java code. Assume conn references a valid Connection object.

```
// This code has an error!!!
String sql = "SELECT * FROM Coffee";
Statement stmt = conn.createStatement();
ResultSet result = stmt.execute(sql);
```

## Algorithm Workbench

1. What SQL data types correspond with the following Java types?

- int
- float
- String
- double

2. Look at the following SQL statement:

```
SELECT Name FROM Employee
```

What is the name of the table from which this statement is retrieving data?

What is the name of the column that is being retrieved?

*For Questions 3 through 11, assume a database has a table named Stock, with the following columns:*

| <b>Column Name</b> | <b>Type</b> |
|--------------------|-------------|
| TradingSymbol      | CHAR(10)    |
| CompanyName        | CHAR(25)    |
| NumShares          | INT         |
| PurchasePrice      | DOUBLE      |
| SellingPrice       | DOUBLE      |

3. Write a SELECT statement that will return all of the columns from every row in table.
4. Write a SELECT statement that will return the TradingSymbol column from every row in table.
5. Write a SELECT statement that will return the TradingSymbol column and the NumShares column from every row in table.
6. Write a SELECT statement that will return the TradingSymbol column only from the rows where PurchasePrice is greater than 25.00.
7. Write a SELECT statement that will return all of the columns from the rows where TradingSymbol starts with “SU”.
8. Write a SELECT statement that will return the TradingSymbol column only from the rows where SellingPrice is greater than PurchasePrice and NumShares is greater than 100.
9. Write a SELECT statement that will return the TradingSymbol column

and the NumShares column only from the rows where SellingPrice is greater than PurchasePrice and NumShares is greater than 100. The results should be sorted by the NumShares column, in ascending order.

10. Write an SQL statement that will insert a new row into the Stock table. The row should have the following column values:

```
TradingSymbol: XYZ
CompanyName: "XYZ Company"
NumShares: 150
PurchasePrice: 12.55
SellingPrice: 22.47
```

11. Write an SQL statement that does the following: For each row in the Stock table, if the TradingSymbol column is “XYZ”, change it to “ABC”.
12. Write an SQL statement that will delete rows in the Stock table where the number of shares is less than 10.
13. Assume the following declaration exists:

```
final String DB_URL = "jdbc:derby:CoffeeDB";
```

The string referenced by DB\_URL is a database URL. Write a statement that uses this string to get a connection to the database.

14. Assuming conn references a valid Connection object, write code to create a Statement object. (Do not be concerned about result set scrolling or concurrency.)
15. Look at the following declaration:

```
String sql = "SELECT * FROM Coffee WHERE Price > 10.00";
```

Assume stmt references a valid Statement object. Write code that executes the SQL statement referenced by the sql variable.

16. Assume the following code is used to retrieve data from the CoffeeDB database’s Coffee table. Write the code that should appear inside the

loop to display the contents of the result set.

```
String sql = "SELECT * FROM Coffee";
Connection conn = DriverManager.getConnection(DB_URL);
Statement stmt = conn.createStatement();
ResultSet result = stmt.executeQuery(sql);
while (result.next())
{
 // Finish this code!!
}
stmt.close();
conn.close();
```

17. Write the SQL statement to create a table named car. The car table should have the columns to hold a car's manufacturer, year model, and a 20-character vehicle ID number.
18. Write an SQL statement to delete the car table you created in Question 17.

## Short Answer

1. If you are writing an application to store the customer and inventory records for a large business, why would you not want to use traditional text or binary files?
2. You hear a fellow classmate say the following: “JDBC is a standard language for working with database management systems. It was invented at IBM.” Are these statements correct, or is he confusing JDBC with something else?
3. When we speak of database organization, we speak of such things as tables, rows, and columns. Describe how the data in a database is organized into these conceptual units.
4. What is a primary key?
5. What is a result set?

6. What are the relational operators in SQL for the following comparisons?
  - Greater than
  - Less than
  - Greater than or equal to
  - Less than or equal to
  - Equal to
  - Not equal to
7. What is the number of the first row in a table? What is the number of the first column in a table?
8. What is metadata? What is result set metadata? When is result set metadata useful?
9. What is a foreign key?

# Programming Challenges

## 1. Customer Inserter

Write an application that connects to the `CoffeeDB` database and allows the user to insert a new row into the `Customer` table.



**VideoNote** The Customer Inserter Problem

## 2. Customer Updater

Write an application that connects to the `CoffeeDB` database and allows the user to select a customer, then change any of that customer's information. (You should not attempt to change the customer number because it is referenced by the `UnpaidOrder` table.)

## 3. Unpaid Order Sum

Write an application that connects to the `CoffeeDB` database, then calculates and displays the total amount owed in unpaid orders. This will be the sum of each row's `Cost` column.

## 4. Unpaid Order Lookup

Write an application that connects to the `CoffeeDB` database and displays a list of customers with unpaid orders. The user should be able to select a customer from the list and see a summary of all the unpaid orders for that customer.

## 5. Population Database

Make sure you have downloaded the book's source code from the companion Web site at [www.pearsonhighered.com/gaddis](http://www.pearsonhighered.com/gaddis). In this chapter's source code folder, you will find a program named

`CreateCityDB.java`. Compile and run the program. The program will create a Java DB database named `cityDB`. The `cityDB` database will have a table named `city`, with the following columns:

| <b>Column Name</b>      | <b>Data Type</b>                          |
|-------------------------|-------------------------------------------|
| <code>CityName</code>   | <code>CHAR (50)</code> <i>Primary key</i> |
| <code>Population</code> | <code>DOUBLE</code>                       |

The `CityName` column stores the name of a city, and the `Population` column stores the population of that city. After you run the `CreateCityDB.java` program, the `city` table will contain 20 rows with various cities and their populations.

Next, write a program that connects to the `cityDB` database and allows the user to select any of the following operations:

- Sort the list of cities by population, in ascending order.
- Sort the list of cities by population, in descending order.
- Sort the list of cities by name.
- Get the total population of all the cities.
- Get the average population of all the cities.
- Get the highest population.
- Get the lowest population.

## 6. Personnel Database Creator

Write an application that creates a database named `Personnel`. The database should have a table named `Employee`, with columns for employee ID, name, position, and hourly pay rate. The employee ID should be the primary key. Insert at least five sample rows of data into the `Employee` table.

## 7. Employee Inserter

Write an application that allows the user to add new employees to the Personnel database you created in Programming Challenge 6.

## 8. Employee Updater

Write an application that allows the user to look up an employee in the Personnel database you created in Programming Challenge 6. The user should be able to change any of the employee's information except employee ID, which is the primary key.

## 9. PhoneBook Database

Write an application that creates a database named PhoneBook. The database should have a table named Entries, with columns for a person's name and phone number.

Next, write an application that lets the user add rows to the Entries table, look up a person's phone number, change a person's phone number, and delete specified rows.

# Appendix A The ASCII/Unicode Characters

The following table lists the first 127 Unicode character codes, which are the same as the ASCII (American Standard Code for Information Interchange) character set. This group of character codes is known as the *Latin Subset of Unicode*. The code columns show character codes and the character columns show the corresponding characters. For example, the code 65 represents the letter A. Note the first 31 codes and code 127 represent control characters that are not printable.

| Code | Character | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0    | NUL       | 26   | SUB       | 52   | 4         | 78   | N         | 104  | h         |
| 1    | SOH       | 27   | Escape    | 53   | 5         | 79   | O         | 105  | i         |
| 2    | STX       | 28   | FS        | 54   | 6         | 80   | P         | 106  | j         |
| 3    | ETX       | 29   | GS        | 55   | 7         | 81   | Q         | 107  | k         |
| 4    | EOT       | 30   | RS        | 56   | 8         | 82   | R         | 108  | l         |
| 5    | ENQ       | 31   | US        | 57   | 9         | 83   | S         | 109  | m         |
| 6    | ACK       | 32   | (Space)   | 58   | :         | 84   | T         | 110  | n         |
| 7    | BEL       | 33   | !         | 59   | ;         | 85   | U         | 111  | o         |
| 8    | Backspace | 34   | "         | 60   | <         | 86   | V         | 112  | p         |
| 9    | HTab      | 35   | #         | 61   | =         | 87   | W         | 113  | q         |
| 10   | Line Feed | 36   | \$        | 62   | >         | 88   | X         | 114  | r         |
| 11   | VTAB      | 37   | %         | 63   | ?         | 89   | Y         | 115  | s         |
| 12   | Form Feed | 38   | &         | 64   | @         | 90   | Z         | 116  | t         |
| 13   | CR        | 39   | '         | 65   | A         | 91   | [         | 117  | u         |
| 14   | SO        | 40   | (         | 66   | B         | 92   | \         | 118  | v         |
| 15   | SI        | 41   | )         | 67   | C         | 93   | ]         | 119  | w         |
| 16   | DLE       | 42   | *         | 68   | D         | 94   | ^         | 120  | x         |
| 17   | DC1       | 43   | +         | 69   | E         | 95   | _         | 121  | y         |
| 18   | DC2       | 44   | ,         | 70   | F         | 96   | `         | 122  | z         |
| 19   | DC3       | 45   | -         | 71   | G         | 97   | a         | 123  | {         |

|    |     |    |   |    |   |     |   |     |   |
|----|-----|----|---|----|---|-----|---|-----|---|
| 20 | DC4 | 46 | . | 72 | H | 98  | b | 124 |   |
| 21 | NAK | 47 | / | 73 | I | 99  | c | 125 | } |
| 22 | SYN | 48 | 0 | 74 | J | 100 | d | 126 | ~ |
| 23 | ETB | 49 | 1 | 75 | K | 101 | e | 127 | D |
| 24 | CAN | 50 | 2 | 76 | L | 102 | f |     |   |
| 25 | EM  | 51 | 3 | 77 | M | 103 | g |     |   |

# Appendix B Operator Precedence and Associativity

This table shows the precedence and associativity of all the Java operators. The table is divided into groups, and each operator in a group has the same precedence. The groups of operators are arranged from the highest precedence at the top of the table, to the lowest precedence at the bottom of the table. For example, the first group of operators shown is:

. [] () ++ --

This group of operators has the highest precedence of all the operators, and each of these operators has the same precedence.

| Operator        | Description          | Associativity |
|-----------------|----------------------|---------------|
| .               | membership           | left-to-right |
| []              | array subscript      | left-to-right |
| ()              | method argument list | left-to-right |
| ++              | postfix increment    | left-to-right |
| --              | postfix decrement    | left-to-right |
| ++              | prefix increment     | right-to-left |
| --              | prefix decrement     | right-to-left |
| +               | unary plus           | right-to-left |
| -               | unary minus          | right-to-left |
| ~               | bitwise complement   | right-to-left |
| !               | logical NOT          | right-to-left |
| new             | object creation      | right-to-left |
| ( <i>type</i> ) | cast                 | right-to-left |
| *               | multiplication       | left-to-right |
| /               | division             | left-to-right |
| %               | remainder            | left-to-right |

|            |                          |               |
|------------|--------------------------|---------------|
| +          | addition                 | left-to-right |
| +          | string concatenation     | left-to-right |
| -          | subtraction              | left-to-right |
| <<         | left shift               | left-to-right |
| >>         | signed right shift       | left-to-right |
| >>>        | unsigned right shift     | left-to-right |
| <          | less than                | left-to-right |
| >          | greater than             | left-to-right |
| <=         | less than or equal to    | left-to-right |
| >=         | greater than or equal to | left-to-right |
| instanceof | type comparison          | left-to-right |
| ==         | equal to                 | left-to-right |
| !=         | not equal to             | left-to-right |
| &          | bitwise AND              | left-to-right |
| ^          | bitwise XOR              | left-to-right |
|            | bitwise OR               | left-to-right |
| &&         | logical AND              | left-to-right |
|            | logical OR               | left-to-right |
| ?:         | conditional              | right-to-left |
| =          | assignment               | right-to-left |
| +=         | combined assignment      | right-to-left |
| -=         | combined assignment      | right-to-left |
| *=         | combined assignment      | right-to-left |
| /=         | combined assignment      | right-to-left |
| <<=        | combined assignment      | right-to-left |
| >>=        | combined assignment      | right-to-left |
| >>>=       | combined assignment      | right-to-left |
| &=         | combined assignment      | right-to-left |
| ^=         | combined assignment      | right-to-left |
| =          | combined assignment      | right-to-left |

# **Index**

# Symbols

- - (subtraction operator), [58](#), [59](#)
- - - (decrement operator), [275–278](#)
- -= (combined assignment), [67–68](#)
- ! (logical NOT operator), [222](#), [227](#)
- != (not equal to operator), [196](#), [197](#)
- % (modulus operator), [58](#), [59](#)
- %= (combined assignment), [67–68](#)
- && (logical AND operator), [222](#), [223–225](#)
- () (parentheses), [33](#), [62](#)
- \* (multiplication operator), [11](#), [58](#), [59](#)
- \*= (combined assignment), [67–68](#)
- , (comma separators), [109–110](#)
- / (division operator), [58](#), [59](#)
- /\* \*/ (comment symbols), [82–83](#)
- /\*\* \*/ (comment symbols), [84](#)
- // (comment symbol), [30](#), [33](#), [81](#)
- /= (combined assignment), [67–68](#)
- ; (semicolon), [11](#), [32–33](#), [200](#)

- \ (backslash), [38](#)
- \\ (backslash escape sequence), [40](#)
- \' (single quote escape sequence), [40](#)
- \" (double quote escape sequence), [40](#)
- \b (backspace escape sequence), [40](#)
- \n (newline escape sequence), [38–40](#)
- \r (return escape sequence), [40](#)
- \t (tab escape sequence), [39, 40](#)
- { (opening braces), [31–33](#)
- ||(OR logical operator), [222, 225–226](#)
- } (closing braces), [31–33](#)
- ' (single quotation marks), [53–54](#)
- " " (double quotation marks), [33, 42](#)
- + (addition operator), [58, 60](#)
- + (string concatenation operator), [43, 388, 561](#)
- ++ (increment operator), [275–278](#)
- += (combined assignment), [67–68](#)
- < (less than operator), [195, 196, 1034](#)
- <= (less than or equal to operator), [195, 196, 1034](#)
- <> (diamond operator), [530](#)

- $\neq$  (not equal to operator), [1034](#)
- $=$  (assignment operator), [11](#), [42](#), [137](#), [197](#)
- $=$  (equal to operator), [1034](#)
- $\equiv$  (equal to operator), [195](#), [196](#), [197](#)
- $>$  (greater than), [195](#), [196](#), [1034](#)
- $\geq$  (greater than or equal to operator), [195](#), [196](#), [1034](#)

# A

- abstract classes, [647–653](#)
- abstract methods, [647–653](#)
- access specification, [147](#)
- access specifier, [30](#)
  - private, [133](#), [134](#), [634](#)
  - protected, [628–634](#)
  - public, [30](#), [133](#), [134](#), [634](#)
  - in UML diagram, [633](#)
- accessor methods, [146](#)
- accumulator variable, [303](#)
- address, [4](#)
- aggregation, [397–410](#)
  - security issues, [405–407](#)
  - in UML, [404–405](#)
- algorithm, [6](#)
- algorithm decomposition, [252–253](#)
- ALU. See [arithmetic and logic unit \(ALU\)](#)
- AND operator (SQL), [1039–1040](#)

- animation, [926–927](#)
  - controlling, [942](#)
  - `FadeTransition` class, [940–941](#)
  - `FillTransition` class, [939–940](#)
  - `RotateTransition` class, [930–934](#)
  - `ScaleTransition` class, [935–937](#)
  - `StrokeTransition` class, [938–939](#)
  - `TranslateTransition` class, [927–930](#)
- Animation class, methods inherited from, [942](#)
- anonymous inner classes, [667–670](#)
  - and event handlers, [789–791](#)
  - and key event handler, [966–967](#)
- API (Application Programmer Interface), [35–36](#)
- Application class, JavaFX, [749–751](#)
- Application Programmer Interface (API), [35–36](#)
- application software, [5](#)
- Arc class, [910–914](#)
- ArcType class, [911–912](#)
- argument, [36](#), [149–151](#)
  - arrays, [468–471](#)

- command-line, [518–519](#)
- `main` methods, [518–519](#)
- objects, [372–385](#)
- passed by value, [151](#)
- passing multiple, [149–151](#)
- variable-length, [519–522](#)
- arithmetic and logic unit (ALU), [3](#)
- arithmetic operators, [58–66](#)
- `ArrayList` class, [522–530](#)
  - add method, [527–528](#)
  - autoboxing and unboxing, [586](#)
  - capacity, [528–529](#)
  - creating an object of, [522–524](#)
  - diamond operator (`<>`), [530](#)
  - enhanced `for` loop method, [524–525](#)
  - get method, [523–524](#)
  - remove method, [525–526](#)
  - size method, [523–524](#)
  - `toString` method, [525](#)
- arrays:

- arguments, as, [468–471](#)
- averaging the values of, [474](#)
- binary search, [501–504](#), [999–1002](#)
- bounds checking, [452–453](#)
- comparing, [472–473](#)
- concept, [445](#)
- content, [456–467](#)
- converting ObservableList to, [853](#)
- copying, [465–467](#)
- declaration, [445–446](#)
- declaration, alternate notation, [455–456](#)
- elements, [447–448](#)
- enhanced for loop, [461–462](#)
- files and, [483–484](#)
- highest value in, finding, [474–475](#)
- initializing ListView control with, [847–848](#)
- initialization, [453–455](#)
- initialization list, [454](#)
- inputting and outputting contents, [448–452](#)
- introduction to, [445–456](#)

- length, [461](#)
- lowest value in, finding, [474–475](#)
- objects, of, [490–494](#)
- off-by-one error, [453](#)
- partially filled, [482–483](#)
- ragged, [516–517](#)
- reference variables, [464–465](#)
- returning from methods, [484–486](#)
- selection sort, [497–500](#)
- sequential search, [494–496](#)
- size declarator, [446–447](#)
- size, user specified, [462–464](#)
- String objects, [486–490](#)
- subscript, [447](#)
- summing contents of, [992–994](#)
- summing values of, [473](#)
- three or more dimensions, [517–518](#)
- two-dimensional, [505–517](#)
- ASCII/Unicode characters, [A-1](#)
- aspect ratio, images, [761–762](#)

- assignment statement, [42](#)
- associativity, operators, [61–62](#), [B-1–B-2](#)
- attributes, [19](#), [129](#), [132–133](#)
- auto commit mode, JDBC, [1085](#)
- autoboxing, [585–586](#)
- AVG function (SQL), [1040–1041](#)

# B

- BankAccount class, [161–167](#)
- base class, [602](#)
- BASIC programming language, [7](#)
- binary digit, [4](#)
- binary files, [317](#), [718–724](#)
  - appending data to, [724](#)
  - reading data from to, [721–723](#)
  - writing data to, [719–721](#)
- binary number, [718–719](#)
  - converting to, [584–585](#)
- binary operator, [58](#)
- binary search algorithm, [501–504](#)
  - recursive version, [999–1002](#)
- binding, [362](#)
- bit, [4](#)
- block, [200–201](#)
- block comments, [83](#), [135](#)
- blueprint metaphor, [131](#)

- boolean data type, [52–53](#)
- BorderPane layout container, [794–798](#)
- BoxBlur class, [944](#), [948](#), [949](#)
- break statement, [316](#)
- buffer:
  - file, [318](#)
  - keyboard, [93](#)
- Button control, [748](#), [749](#), [778–784](#)
  - and events handling, [780–784](#)
- byte, [4](#)
- byte code, [8](#), [12–13](#)
- Byte class, [583](#)
  - parseByte method, [584](#)
  - toString method, [584](#)
- byte data type, [48–49](#), [71](#)

# C

- C programming languages, [7](#)
- C# programming languages, [7](#)
- C++ programming languages, [7](#)
- call stack, [705](#)
- car class, [181](#)
- Cascading Style Sheet (CSS)
  - applying to JavaFX application, [812–814](#)
  - RGB color system, [818–820](#)
  - style properties, [811–812](#)
  - styling JavaFX applications with, [809–823](#)
  - type selectors, JavaFX nodes and corresponding, [810–811](#)
- case key word, [239–245](#)
- case-sensitive language, [31](#), [33](#)
- cast operators, [69–71](#)
- catch clause, [690](#), [696](#)
  - multiple exceptions, handling, [706–708](#)
- CD (compact disc), [5](#)
- CellPhone class, [168–171](#)

- central processing unit (CPU), [2](#), [3–4](#)
- char data type, [53–55](#)
- character(s), [53–55](#)
  - comparing, [201–202](#)
  - conversion, [549–550](#)
  - literals, [53–54](#)
  - reading from keyboard, [92](#)
  - testing, [544–548](#)
- Character class, [544–550](#)
  - `isDigit` method, [544](#)
  - `isLetter` method, [544](#)
  - `isLetterOrDigit` method, [544](#)
  - `isLowerCase` method, [544](#)
  - `isSpaceChar` method, [544](#)
  - `isUpperCase` method, [544](#)
  - `isWhiteSpace` method, [544](#)
  - `toLowerCase` method, [549](#)
  - `toUpperCase` method, [549](#)
- CHARACTER/CHAR data type (SQL), [1021](#)
- check menu item, [874](#)

- CheckBox controls, [834–839](#)
  - clicks, responding to, [839](#)
  - determination of selection, [834–835](#)
  - selection in code, [835–839](#)
- checked exceptions, [709–710](#)
- CheckMenuItem class, [874](#)
- Circle class, constructors, [900–903](#)
- class, [21–22](#), [129–149](#)
  - abstract, [647–653](#)
  - adapter, [959](#)
  - body, [31](#)
  - building, step by step, [132–134](#)
  - collaboration, [427–431](#)
  - constructors, [155–161](#)
  - constructors, copy, [396–397](#)
  - constructors, default, [159](#), [618](#)
  - constructors, overloaded, [365–371](#)
  - definition, [30](#)
  - exceptions, [688–689](#), [714–717](#)
  - field, [129](#), [134](#)

- finding, [174–182](#)
- header, [30](#)
- hierarchy, [640](#)
- inheritance from subclasses, [635–640](#)
- inner, [413–416](#)
- instance field, [152–155](#)
- instance method, [152–155](#)
- instance of, [22](#), [75](#), [130–132](#)
- key word, [30](#)
- layout of members, [148](#)
- members, [133](#)
- names, [46](#)
- no-arg constructor, [160](#), [618](#)
- Object, [640–642](#)
- overloaded methods, [360–365](#)
- protected members, [628–634](#)
- static fields, [354–356](#)
- static members, [353–359](#)
- static methods, [354](#), [356–359](#)
- type variable, [75–76](#)

- wrapper, [543–544](#)
- class file extension, [14](#)
- class-type variables, [75–76](#)
- close() method
  - DataInputStream class, [722](#)
  - DataOutputStream class, [720](#)
- COBOL programming languages, [7](#)
- collaboration, class, [427–431](#)
- ColorAdjust class, [944](#), [948](#)
- colors
  - in CSS, [818–820](#)
  - stroke, changing, [900](#)
- columns, DBMS, [1019–1022](#)
  - content retrieval, ResultSet object and, [1026–1027](#)
- combined assignment operators, [67–68](#)
  - %=, [67–68](#)
  - \*=, [67–68](#)
  - +=, [67–68](#)
  - -=, [67–68](#)
  - /=, [67–68](#)

- ComboBox class
  - `getValue()` method, [861–863](#)
  - list of methods, [861](#)
- ComboBox controls, [860–865](#)
  - `editable`, [865](#)
  - responding to item selection with event handler, [863–864](#)
  - retrieving selected item, [861–863](#)
- comma (,) separators, [109–110](#)
- comma separated value file format, [587](#)
- command-line arguments, [518–519](#)
- command line interface, [745](#)
- comments, [30](#)
  - `/* */` symbols, [82–83](#)
  - `/** */` symbols, [84](#)
  - `//` symbols, [81](#)
  - documentation, [83–85](#)
  - multiline, [82–83](#)
  - single-line, [81–82](#)
- commit, transaction-processing code, [1085](#)
- compact disc, [5](#)

- compilers, [12](#)
- compiling and running a program, [14](#)
- component reusability, of OOP, [20](#)
- compound border, [811](#)
- compound operator, [67](#)
- concatenation, [43](#), [388](#), [561](#)
- conditional loop, [292](#)
- conditional operator, [237–238](#)
- conditionally-executed code, [195](#), [199–201](#), [212](#), [218](#)
- Connection class
  - commit method, [1085](#)
  - createStatement method, [1024](#), [1045](#), [1048](#), [1053](#), [1061](#)
  - rollback method, [1085](#)
  - setAutoCommit method, [1085](#)
- console, [35](#)
- console interface, [745](#)
- console output, [35](#)
- constants
  - enum, [756](#)
  - Interpolator, [943](#)

- constants, enum, [420](#)
- constants, named, [72–74](#)
- constructors, [155–161](#)
  - Arc class, [910](#), [914](#)
  - Circle class, [900–903](#)
  - copy, [396–397](#)
  - default, [159](#), [367–368](#), [618](#)
  - FadeTransition class, [940](#)
  - FileOutputStream class, [724](#)
  - FillTransition class, [939](#)
  - Line class, [897](#)
  - no-arg, [160](#), [618](#)
  - overloaded, [365–371](#)
  - Polyline class, [917](#)
  - RandomAccessFile class, [725](#)
  - Rectangle class, [904–907](#)
  - RotateTransition class, [930](#)
  - ScaleTransition class, [935](#)
  - StrokeTransition class, [938](#)
  - superclass, [610–619](#)

- Text class, [919](#)
- TranslateTransition class, [927](#)
- UML in, [160](#)
- continue statement, [316](#)
- control character, [38](#)
- control unit, [3](#)
- controller class, JavaFX
- controls, JavaFX
  - Button control, [748](#), [749](#), [778–784](#)
  - CheckBox controls, [834–839](#)
  - ComboBox controls, [860–865](#)
  - creating, [752](#)
  - Label control, [748](#), [749](#)
  - ListView control, [839–860](#)
  - RadioButton controls, [824–833](#)
  - Slider controls, [866–871](#)
  - TextArea controls, [871–873](#)
  - TextField control, [748](#), [749](#), [785–789](#)
- conversion, data type, [68–72](#)
  - strings to numbers, [99–102](#)

- COUNT function (SQL), [1041](#)
- count-controlled loop, [292–293](#)
- CPU. See [central processing unit \(CPU\)](#)
- CRC cards, [430–431](#)
- CREATE TABLE statement, [1056–1059](#)
- custom style classes, creating, [820–822](#)
- Customer class, [180–181](#)

# D

- data hiding, [19–20](#), [146–147](#)
- data type, [47–57](#)
  - boolean, [52–53](#)
  - byte, [48–49](#), [71](#)
  - char, [53–55](#)
  - conversion, [68–72](#)
  - float, [50](#), [51](#)
  - floating-point, [50–52](#)
  - int, [48–49](#), [71](#)
  - integer, [48–50](#)
  - long, [48–49](#)
  - primitive, [47–57](#)
  - ranking, [69](#)
  - short, [48–49](#), [71](#)
  - UML, [147–149](#)
- database management system (DBMS), [1013–1022](#)
  - columns, [1019–1022](#)
  - Java DB, [1016](#)

- JDBC, [1014–1015](#)
  - password-protected database, connecting, [1019](#)
  - primary keys, [1022](#)
  - rows, [1019–1021](#)
  - SQL, [1015](#)
  - SQL statement passing to, [1024–1034](#)
  - stored procedures, [1086](#)
  - tables, [1019–1022](#)
  - usage, [1015–1016](#)
- DataInputStream class, [721–722](#)
    - close methods, [722](#)
    - readBoolean method, [722](#)
    - readByte method, [722](#)
    - readChar method, [722](#)
    - readDouble method, [722](#)
    - readFloat method, [722](#)
    - readInt method, [722](#)
    - readLong method, [722](#)
    - readShort method, [722](#)
    - readUTF method, [723–724](#)

- `DataOutputStream` class, [719–720](#)
  - `close` method, [720](#)
  - `writeBoolean` method, [720](#)
  - `writeByte` method, [720](#)
  - `writeChar` method, [720](#)
  - `writeDouble` method, [720](#)
  - `writeFloat` method, [720](#)
  - `writeInt` method, [720](#)
  - `writeLong` method, [720](#)
  - `writeShort` method, [720](#)
  - `writeUTF` method, [720](#), [723–724](#)
- `DATE` data type (SQL), [1021](#), [1070](#)
- `DBViewer` application, [1066](#)
- `DECIMAL` data type (SQL), [1021](#)
- decision structure, [194–195](#)
- declaration, variable, [42](#), [48–49](#)
- decomposition, algorithm, [252–253](#)
- decorations, window, [754](#)
- decrement operator (`--`), [275–278](#)
  - postfix mode, [277–278](#)

- prefix mode, [277–278](#)
- default methods, interfaces, [660–662](#)
- deep copy, [405](#), [406](#)
- default, [159](#), [367–368](#), [618](#)
- default exception handler, [688](#)
- DELETE statement, [1052–1053](#)
- delimiter, [319](#), [575](#)
  - multiple, using, [579–580](#)
- depth, of recursion, [987](#)
- derived class, [602](#)
- deserialization, [731](#)
- design, object-oriented, [174–182](#), [427–431](#)
- dialog boxes, [95–102](#)
  - input dialog, [95–97](#)
  - message dialog, [95–96](#)
- diamond operator `<>`, [530](#)
- direct recursion, [992](#)
- discounts, calculating, [63–65](#)
- disk drive, [4](#)
- division, integer, [60–61](#)

- division by zero, [203–204](#)
- documentation comments, [83–85](#)
- Double class, [583](#)
  - MAX\_VALUE constant, [585](#)
  - MIN\_VALUE constant, [585](#)
  - parseDouble method, [584](#)
  - toString method, [584](#)
- DOUBLE data type (SQL), [1021](#)
- double data type, [48](#), [50–52](#)
- do-while loop, [289–292](#)
- DriverManager.getConnection method, [1019](#)
- DropShadow class, [944–947](#)
  - methods, [946](#)
- DROP TABLE statement, [1059](#)
- DVD drive, [5](#)
- dynamic binding, polymorphism and, [643–644](#)

# E

- E notation, [51–52](#)
- elements, of arrays, [447–448](#)
- Ellipse class, constructors, [907–910](#)
- else clause, [204](#)
  - trailing, [218](#)
- encapsulation, [19](#)
- EndOfMedia event handlers, [957–960](#)
- entity relationship diagrams, [1071](#)
- enum constants, [416, 756](#)
- enum key word, [416](#)
- enumerated types, [416–424](#)
  - compareTo method, [418–419](#)
  - equals method, [418](#)
  - ordinal method, [418](#)
  - ordinal value, [418](#)
  - switching on, [423–424](#)
- EOFException class, [689, 722](#)
- equals method, [418, 641–642](#)

- Error class, [689](#), [709](#)
- errors
  - logical, [17](#)
  - syntax, [12](#), [14](#)
- escape sequence, [38–40](#)
  - \\", [38](#)
  - \' , [40](#)
  - \" , [40](#)
  - \b , [40](#)
  - \n , [38–40](#)
  - \r , [40](#)
  - \t , [39](#), [40](#)
  - control character, [38](#)
- event handlers, [780](#)
  - anonymous inner classes and, [789–791](#)
  - EndOfMedia, [957–960](#)
  - key events, [965–971](#)
  - lambda expressions and, [792–794](#)
  - mouse events, [972–977](#)
  - registering, [782–784](#)

- responding to `ComboBox` item selection with, [863–864](#)
- responding to `ListView` item selection with, [845–846](#)
- writing, [781](#)
- event-driven GUI programs, [747](#)
- events
  - definition, [780](#)
  - firing, [780](#)
  - handling, with `Button` control, [780–784](#)
  - object, [780](#)
  - source, [780](#)
- Excel spreadsheet, [587](#)
- Exception class, [689](#), [696](#), [711](#)
- exceptions, [320](#), [687–717](#)
  - checked, [709–710](#)
  - classes, [688–689](#), [714–717](#)
  - default error message for, [694–696](#)
  - default handler, [688](#)
  - defined, [688](#)
  - handler, [688](#)
  - handling, [688](#), [690–693](#)

- multiple, handling, [696–698](#), [706–708](#)
- polymorphic references to, [696](#)
- throw statement, [710–711](#)
- throwing, [710–717](#)
- unchecked, [709](#)
- when not caught, [708](#)
- executable file, [12](#)
- explicit import statements, [173](#)
- extends key word, [607](#), [640](#)
- external drives, [5](#)

# F

- factorial algorithm, [989–992](#)
- `FadeTransition` class, [940–941](#)
- fetch/decode/execute cycle, [4](#)
- Fibonacci series, [996–997](#)
- field, [129](#), [134](#), [172](#)
  - instance, [152–155](#), [354–356](#)
- `FileChooser` class, [883–884](#)
  - `showOpenDialog` method, [884](#)
  - `showSaveDialog` method, [884](#)
- `FileInputStream` class, [721](#), [731](#)
- `FileNotFoundException` class, [689](#), [692](#), [696](#), [701](#), [709](#), [725](#)
- `FileOutputStream` class, [719](#), [731](#)
  - constructor, [724](#)
- files
  - appending data to, [323–324](#)
  - binary, [317](#), [718–724](#)
  - checking for existence, [333–336](#)
  - closing, [317](#)

- detecting the end of, [328–330](#)
- input, [317](#)
- location, specifying, [324](#)
- opening, [317](#)
- output, [317](#)
- pointer, [727–730](#)
- random access, [724–730](#)
- read position, [326](#)
- reading data from, [325–333](#)
- reading primitive values from, [330–332](#)
- sequential access, [724](#)
- text, [317](#)
- `FillTransition` class, [939–940](#)
- `final` key word, [72–73](#), [628](#)
- `finally` clause, [703–704](#)
- flags, [109–112](#). See also `numbers`
  - `if` statement, [201](#)
- flash memory, [5](#)
- `Float` class, [583](#)
  - `MAX_VALUE` constant, [585](#)

- MIN\_VALUE constant, [585](#)
- parseFloat method, [584](#)
- toString method, [584](#)
- for loop, [292–302](#)
  - counter variable, [294](#), [296](#)
  - initialization expression, [293](#), [297](#), [299–300](#)
  - pretest, [296](#)
  - test expression, [293–294](#)
  - update expression, [293](#), [296–297](#), [299–300](#)
  - user controlled, [297–299](#)
- foreign key, [1069](#)
- format specifier
  - flags, [109–112](#)
  - minimum field width, [106–108](#)
  - precision, [106](#), [107–108](#)
  - string argument, [112–116](#)
  - syntax for, [105](#)
- FORTRAN programming languages, [7](#)
- functional interface, [670–673](#)
- functional RGB notation, [819](#)

# G

- garbage collection, [425–426](#)
  - finalize method, [426](#)
- “garbage in, garbage out,” [286](#)
- GaussianBlur class, [944](#), [948](#), [950](#)
- generalization and specialization, [601–602](#)
- getArea method, [144–146](#)
- getLength method, [140–143](#)
- getMessage method, [694–696](#), [710](#), [712](#)
- getSelectionModel().getSelectedIndex() method, [842–844](#), [849](#)
- getSelectionModel().getSelectedItem() method, [841–842](#), [849](#)
- getWidth method, [140–143](#)
- Glow class, [944](#), [952](#)
- Gosling, James, [8](#)
- graphical user interface (GUI), [35](#), [745–747](#)
  - application, debugging, [884–888](#)
- greatest common divisor, finding, [998–999](#)
- Green Team, [8](#)
- GridPane layout container, [770–776](#)

- GUI (graphical user interface), [35](#), [745–747](#)
  - application, debugging, [884–888](#)

# H

- hard drive, [4–5](#)
- hardware
  - central processing unit (CPU), [2, 3–4](#)
  - components, [2, 3](#)
  - defined, [2](#)
  - input devices, [2, 5](#)
  - main memory, [2, 4](#)
  - output devices, [2, 5](#)
  - secondary storage, [2, 4–5](#)
- “has a” relationship, [402](#)
- HBox layout container, [753, 763–768](#)
- header
  - class, [30](#)
  - loop, [279, 293](#)
  - method, [31, 135–136](#)
- heavyweight components, [741](#)
- hex numbers, converting to, [584–585](#)
- hierarchy, class, [640](#)

- HTML. See [Hypertext Markup Language \(HTML\)](#)

# I

- ID selector, [822](#)
- IDE. See [integrated development environments \(IDE\)](#)
- identifiers, [9](#), [45–46](#)
- if statement, [193–202](#)
  - comparing characters, [201–202](#)
  - conditionally executed statements, [200–201](#)
  - flags, [201](#)
  - nested, [209–216](#)
  - programming style, [199–200](#)
  - relational operators, [195–197](#)
  - semicolons, [200](#)
- if-else statement, [202–204](#)
- if-else-if statement, [217–220](#)
- IllegalArgumentException class, [703](#), [712](#)
- Image class, [758–762](#)
  - setPreserveRatio method, [761–762](#)
- images
  - aspect ratio, preserving, [761–762](#)

- displaying, [758–762](#)
- loading from Internet, [761](#)
- size of, [761](#)
- ImageView class, [759–762](#)
  - setImage method, [762](#)
- implements key word, [654](#), [659](#)
- import statement, [90](#), [173–174](#), [689](#)
- increment operator (++), [275–278](#)
  - postfix mode, [277–278](#)
  - prefix mode, [277–278](#)
- indirect recursion, [992](#)
- inheritance, [22–23](#), [601–612](#)
  - and “is a” relationship, [602](#)
  - chain of, [635–640](#)
  - constructors, superclass, [610–619](#)
  - generalization and specialization, [601–602](#)
  - in UML diagram, [609](#), [639](#)
  - interface, [665](#)
  - subclass, [602](#)
  - superclass, [602](#)

- initialization, [55–56](#)
  - variables, [55–56](#)
- inline style rules, [823](#)
- inner class, [413–416](#), [667](#)
- InnerShadow class, [944](#), [947](#)
- input devices, [2](#), [5](#)
- InputMismatchException class, [696](#), [701](#), [707](#)
- INSERT statement, [1044–1047](#)
  - format of, [1045](#)
  - with JDBC, [1045–1046](#)
- instance, of class, [22](#)
- instanceof operator, [645–646](#)
- Integer class, [583](#)
  - MAX\_VALUE constant, [585](#)
  - MIN\_VALUE constant, [585](#)
  - parseInt method, [584](#)
  - toBinaryString method, [584–585](#)
  - toHexString method, [584–585](#)
  - toOctalString method, [584–585](#)
  - toString method, [584](#)

- `INTEGER` data type (SQL), [1021](#)
- integer division, [60–61](#)
- integrated development environments (IDE), [14](#), [15](#)
- `interface` key word, [653](#)
- interfaces, [653–667](#)
  - as contract, [655–659](#)
  - default methods, [660–662](#)
  - fields in, [659](#)
  - functional, [670–673](#)
  - implementing, [654](#)
  - in UML, [660](#)
  - inheritance, [665](#)
  - multiple, implementing, [659–660](#)
  - `ObservableList`, [799–800](#)
  - polymorphism and, [662–667](#)
  - `Serializable`, [730](#)
- `InventoryItem` class, [368–371](#)
- `IIOException` class, [689](#), [707](#), [725](#)
- “is a” relationship, [602](#), [645](#)
- `isSelected` method, [825](#)

# J

- Java DB, [1016](#)
- Java programming language
  - defined, [7](#)
  - history, [8](#)
  - key words, [9–10](#)
  - overview, [1](#)
- Java Virtual Machine (JVM), [12–13](#)
- JavaFX, [748](#)
  - Application class, [749–751](#)
  - applications styling with CSS, [809–823](#)
  - BoxBlur class, [944](#), [948](#), [949](#)
  - ColorAdjust class, [944](#), [948](#)
  - controls (see [controls, JavaFX](#))
  - custom style classes, creating, [820–822](#)
  - DropShadow class, [944–947](#)
  - FileChooser class, [883–884](#)
  - GaussianBlur class, [944](#), [948](#), [950](#)
  - Glow class, [944](#), [952](#)

- `Image` class, [758–762](#)
  - images, displaying, [758–762](#)
  - `ImageView` class, [759–762](#)
  - `InnerShadow` class, [944, 947](#)
  - layout containers (See [layout containers](#))
  - `Line` class, [897–900](#)
  - `Media` class, [955–960](#)
  - `MediaPlayer` class, [955–960](#)
  - `MediaView` class, [960–965](#)
  - menus, [873–882](#)
  - `MotionBlur` class, [944, 948, 950–951](#)
  - nodes, and corresponding CSS type selectors, [810–811](#)
  - `ObservableList` interface, [799–800](#)
  - `Reflection` class, [944, 953](#)
  - scenes (See [scenes, JavaFX](#))
  - `SepiaTone` class, [944, 951–952](#)
  - `Shape` class, [896–897](#)
  - stages, [749, 751, 754–756](#)
  - transition classes (See [transition classes](#))
- `java.applet` package, [174](#)

- `java.awt` package, [174](#)
- `java.io` package, [174](#), [689](#), [725](#), [730](#)
- `java.lang` package, [640](#), [689](#)
- `java.lang` packages, [174](#)
- `java.net` package, [174](#), [973](#)
- `java.security` package, [174](#)
- `java.sql` package, [174](#), [1024](#)
- `java.swing` package, [174](#)
- `java.text` package, [174](#)
- `java.util` package, [174](#), [696](#)
- JavaScript programming languages, [7](#)
- JDBC (Java Database Connectivity), [1014–1015](#)
  - auto commit mode, [1085](#)
  - deleting rows with, [1053](#)
  - inserting rows with, [1045–1046](#)
  - new database creation with, [1060–1061](#)
  - updating rows with, [1048–1052](#)
- `JOptionPane` class, [95–102](#)
  - input dialog, [95–97](#)
  - message dialog, [95–96](#)

- `showInputDialog` method, [97](#), [99](#)
- `showMessageDialog` method, [96](#)
- JVM. See [Java Virtual Machine \(JVM\)](#)

# K

- key events, [965](#)
- *KEY\_PRESSED* event handler, [966–971](#)
- KEY\_RELEASED, [965–971](#)
- KEY\_TYPED, [965–971](#)
- key words, [9](#), [9–10](#), [45](#)
- keyboard buffer, [93](#)
- keyboard input, reading, [87–95](#)
- KeyEvent class, [965–966](#)

# L

- Label control, [748](#), [749](#)
- lambda expressions, [670–673](#)
  - and event handlers, [792–794](#)
- lambda operator, [671](#)
- late binding, [643–644](#)
- layout containers, [752–753](#), [762](#)
  - BorderPane, [794–798](#)
  - GridPane, [770–776](#)
  - HBox, [753](#), [763–768](#)
  - multiple, in same screen, [777–778](#)
  - VBox, [768–770](#)
- leading zeros, padding numbers with, [110–111](#)
- left to right associativity, [61–62](#)
- left-justified numbers, [112](#)
- lexicographical comparison, [233](#)
- LIKE operator (SQL), [1038–1039](#)
- Line class, [897–900](#)
  - constructors, [897](#)

- `ListView` class
  - `getItems()`.`addAll()` method, [846–847](#)
  - `getItems()`.`setAll()` method, [846–847](#)
  - `getSelectionModel()`.`getSelectedIndex()` method, [842–844](#), [849](#)
  - `getSelectionModel()`.`getSelectedItem()` method, [841–842](#), [849](#)
  - methods to select items, [854](#)
- `ListView` control, [839–860](#)
  - adding items vs. setting items, [846–847](#)
  - change event, [845–846](#)
  - initializing with array/`ArrayList`, [847–848](#)
  - methods to select items, [854](#)
  - orientation, [854](#)
  - preferred size of, [840](#)
  - retrieving multiple selected items, [849–852](#)
  - retrieving selected item, [841–842](#)
  - selected item, retrieving index of, [842–844](#)
  - selection modes, [848–849](#)
- literal(s), [44–45](#)
  - character, [53–54](#)
  - floating-point, [51](#)

- integer, [50](#)
  - string, [32](#), [74](#)
- local variable, [80](#), [172](#)
- logical errors, [17](#)
- logical operators, [222–229](#), [1039–1040](#)
  - ! (NOT), [222](#), [227](#)
  - && (AND), [222](#), [223–225](#)
  - || (OR), [222](#), [225–226](#)
  - associativity, [227–228](#)
  - checking numeric ranges, [228–229](#)
  - precedence, [227–228](#)
  - short-circuit evaluation, [223](#)
- Long class, [583](#)
  - MAX\_VALUE constant, [585](#)
  - MIN\_VALUE constant, [585](#)
  - parseLong method, [584](#)
  - toBinaryString method, [584–585](#)
  - toHexString method, [584–585](#)
  - toOctalString method, [584–585](#)
  - toString method, [584](#)

- loop
  - arrays, with, [461–462](#)
  - body, [280](#)
  - control variable, [282](#)
  - count-controlled, [292–293](#)
  - deciding which to use, [316](#)
  - defined, [279](#)
  - do-while, [289–292](#)
  - for, [292–302](#)
  - header, [279, 293](#)
  - infinite, [282–283](#)
  - iteration, [281](#)
  - nested, [308–315](#)
  - pretest, [282](#)
  - programming style, [283–284](#)
  - user-controlled, [292](#)
  - while, [279–289](#)

# M

- main memory, [2](#), [4](#)
- many to one relationship, [1071](#)
- Math class, [65–66](#), [359](#)
  - PI constant, [73–74](#)
  - pow method, [65–66](#)
  - sqrt method, [66](#)
- MAX function (SQL), [1041](#)
- MAX\_VALUE constant, [585](#)
- Media class, [955–960](#)
- MediaPlayer class, [955–960](#)
  - methods, [956](#)
- MediaView class, [960–965](#)
- meta data, [1063–1069](#)
- method(s), [19](#), [31–32](#), [129](#)
  - accessor, [146](#)
  - abstract, [647–653](#)
  - binding, [643–644](#)
  - DataInputStream class, [722](#)

- DataOutputStream class, [720](#)
- equals, [641–642](#)
- final, [628](#)
- getMessage, [694–696](#), [710](#), [712](#)
- header, [31](#)
- main. See [main method](#)
- mutator, [146](#)
- overloading vs. overriding, [625–627](#)
- overriding, [620–628](#)
- recursive. See [recursion](#)
- ResultSet class, [1062](#)
- signature, [623](#)
- toString, [641–642](#)
- menu bar, [874](#)
- Menu class, [874](#)
- menu item, [874](#)
- MenuBar class, [874](#)
- MenuItem class, [874](#)
- menus
  - components of, [874](#)

- JavaFX, [873–882](#)
- Metric class, [359](#)
- MIN function (SQL), [1041](#)
- MIN\_VALUE constant, [585](#)
- minimum field width, [106–108](#)
- mnemonics, [881–882](#)
- MotionBlur class, [944, 948, 950–951](#)
- mouse events, [972–977](#)
- MouseEvent class, [972–977](#)
- multi-catch, [707](#)
- multiline comments, [82–83](#)
- multithreaded application, [574](#)
- mutator methods, [146](#)

# N

- named colors, [819–820](#)
- named constants, [72–74](#)
- names, programmer-defined, [10](#), [45–46](#)
- naming rules, [45–46](#)
- narrowing conversion, [69](#)
- negation operator, [58](#)
- nested if statement, [209–216](#)
- nested if-else statement, [220](#)
- nested loops, [308–315](#)
- new key word, [137](#)
- newline character, [93–95](#), [319](#)
- nextLine method, [325–328](#)
- no-arg constructors, [160](#), [618](#)
- Node class, [922–924](#)
- nodes, JavaFX
  - BoxBlur class, [944](#), [948](#), [949](#)
  - ColorAdjust class, [944](#), [948](#)
  - combining effects, [953–955](#)

- CSS type selectors, [810–811](#)
- DropShadow class, [944–947](#)
- GaussianBlur class, [944](#), [948](#), [950](#)
- Glow class, [944](#), [952](#)
- ID, [822](#)
- InnerShadow class, [944](#), [947](#)
- MotionBlur class, [944](#), [948](#), [950–951](#)
- Reflection class, [944](#), [953](#)
- root, applying styles to, [816–817](#)
- rotating, [922–924](#)
- scaling, [924–925](#)
- SepiaTone class, [944](#), [951–952](#)
- style properties, [811–812](#)
- nouns, [175–180](#)
- null references, [407–410](#)
- null statement, [200](#)
- NumberFormatException class, [695–696](#), [702–703](#), [707](#)
- numbers
  - left-justified, [112](#)
  - padding, with leading zeros, [110–111](#)

- right-justified, [112](#)
- with comma separators, [109–110](#)

# O

- Object class, [640–642](#)
  - equals method, [641–642](#)
  - toString method, [641–642](#)
- object-oriented design, [174–182](#), [427–431](#)
  - class collaboration, [427–431](#)
  - finding the classes, [174–182](#)
  - problem domain, [175–180](#)
  - responsibilities, identifying, [180–182](#)
- object-oriented programming (OOP), [19–23](#)
  - attributes, [19](#)
  - component reusability, [20](#)
  - data hiding, [19–20](#)
  - encapsulation, [19](#)
  - inheritance, [22–23](#). See [inheritance](#)
  - methods, [19](#)
- ObjectOutputStream class, [731](#)
- objects, [19](#), [21–22](#)
  - as arguments, [372–385](#)

- copy, [394–397](#)
- defined, [19](#)
- everyday example of, [21](#)
- returning from methods, [385–388](#)
- serialization, [730–734](#)
- ObservableList
  - converting to array, [853](#)
  - elements of, [852–853](#)
- ObservableList interface, [799–800](#)
- octal numbers, converting to, [584–585](#)
- off-by-one error, [453](#)
- one to many relationship, [1071](#)
- OOP. See [object-oriented programming \(OOP\)](#)
- operands, [9, 10](#)
- operating system, [5](#)
- operators, [9, 10](#)
  - AND, [1039–1040](#)
  - - (subtraction operator), [58, 59](#)
  - - - (decrement operator), [275–278](#)
  - -= (combined assignment), [67–68](#)

- ! (logical NOT operator), [222](#), [227](#)
- != (not equal to operator), [196](#), [197](#)
- % (modulus operator), [58](#), [59](#)
- %= (combined assignment), [67–68](#)
- && (logical AND operator), [222](#), [223–225](#)
- () (parentheses), [33](#), [62](#)
- \* (multiplication operator), [10](#), [58](#), [59](#)
- \*= (combined assignment), [67–68](#)
- , (comma separators), [109–110](#)
- / (division operator), [58](#), [59](#)
- /= (combined assignment), [67–68](#)
- || (OR logical operator), [222](#), [225–226](#)
- + (addition operator), [58](#), [60](#)
- + (string concatenation operator), [43](#), [388](#), [561](#)
- ++ (increment operator), [275–278](#)
- += (combined assignment), [67–68](#)
- < (less than operator), [195](#), [196](#), [1034](#)
- <= (less than or equal to operator), [195](#), [196](#), [1034](#)
- <> (diamond operator), [530](#)
- <> (not equal to operator), [1034](#)

- = (assignment operator), [10](#), [42](#), [137](#), [197](#)
- = (equal to operator), [1034](#)
- == (equal to operator), [195](#), [196](#), [197](#)
- > (greater than operator), [195](#), [196](#), [1034](#)
- >= (greater than or equal to operator), [195](#), [196](#), [1034](#)
- arithmetic, [58–66](#)
- associativity, [61–62](#), [227–228](#)
- binary, [58](#)
- cast, [69–71](#)
- combined assignment, [67–68](#)
- compound, [67](#)
- conditional, [237–238](#)
- `instanceof`, [645–646](#)
- lambda, [671](#)
- `LIKE`, [1038–1039](#)
- logical, [222–229](#)
- mixed, in expressions, [71–72](#)
- negation, [58](#)
- OR, [1039–1040](#)
- precedence, [61–62](#), [227–228](#)

- precedence and associativity, [B-1–B-2](#)
- relational, [195–197](#)
- ternary, [58](#)
- unary, [58](#)
- OR operator (SQL), [1039–1040](#)
- output devices, [2](#), [5](#)
- overhead, [989](#)
- overloaded methods, [360–365](#)
  - constructors, [365–371](#)
- overloading, methods
  - overriding vs., [625–627](#)
- overriding methods, [620–628](#)
  - overloading vs., [625–627](#)
  - preventing, [628](#)

# P

- package access, [633–634](#)
- packages, [173–174](#)
  - `java.applet`, [174](#), [970](#)
  - `java.awt`, [174](#)
  - `java.io`, [174](#), [689](#), [725](#), [730](#)
  - `java.lang`, [174](#), [640](#), [689](#)
  - `java.net`, [174](#)
  - `java.security`, [174](#)
  - `java.sql`, [174](#), [1024](#)
  - `java.swing`, [174](#)
  - `java.text`, [174](#)
  - `java.util`, [174](#), [696](#)
- padding, `HBox` layout container, [766–768](#)
- padding numbers, with leading zeros, [110–111](#)
- parameter variable, [135–136](#), [138–139](#), [149–151](#), [172](#)
  - UML, in, [147–149](#)
- parse methods, [583–584](#)
- Pascal programming languages, [7](#)

- pass by value, [151](#)
- password-protected database, connecting, [1019](#)
- Payroll class, [205–208](#)
- percentages, calculating, [63–65](#)
- Perl programming languages, [7](#)
- PHP programming languages, [7](#)
- Polygon class, [914–917](#)
- Polyline class, [917–919](#)
- polymorphism, [642–646](#)
  - and dynamic binding, [643–644](#)
  - and exceptions, [696](#)
  - and interfaces, [662–667](#)
- portability, [12](#)
- postfix mode, [277–278](#)
- precedence, operator, [61–62](#), [B-1–B-2](#)
- precision, [106](#), [107–108](#)
- preferred size, scene, [756](#)
- prefix mode, [277–278](#)
- primary keys, [1022](#), [1057](#), [1071](#)
- priming read, [287](#)

- primitive data type, [47–57](#)
- primitive-type variables, [75](#)
- print method, [35–40](#)
- printf method, [102–104](#)
- println method, [32](#), [35–40](#)
- PrintWriter class
  - print method, [319](#)
  - println method, [318–319](#)
- private access specifier, [134](#), [252–253](#), [634](#)
- private methods, [21](#)
- problem domain, [175–180](#)
- procedural programming, [19](#)
- procedure, [19](#)
- programmer-defined names, [10](#), [45–46](#)
- programming languages, [6–8](#)
  - BASIC, [7](#)
  - C, [7](#)
  - C#, [7](#)
  - C++, [7](#)
  - COBOL, [7](#)

- elements, [8–11](#)
- FORTRAN, [7](#)
- Java. See [Java programming language](#)
- JavaScript, [7](#)
- Pascal, [7](#)
- Perl, [7](#)
- PHP, [7](#)
- Python, [7](#)
- Ruby, [7](#)
- Visual Basic, [7](#)
- programming process, [16–18](#)
- programming style, [85–87](#), [199–200](#), [283–284](#)
- protected access specifier, [628–634](#)
- pseudocode, [17](#)
- public access specifier, [30](#), [133](#), [135](#), [634](#)
- public methods, [21](#)
- punctuation, [9](#), [10–11](#)
- Python programming languages, [7](#)

# Q

- queries, [1015](#)
- quotation marks, double (“”), [33](#), [44](#)
- quotation marks, single (‘), [53–54](#)

# R

- radio menu item, [874](#)
- RadioButton class
  - isSelected method, [825](#)
  - setSelected method, [825–830](#)
- RadioButton controls, [824–833](#)
  - action event, responding to, [830–833](#)
  - determination of selection, [825](#)
  - selection in code, [825–830](#)
- RadioMenuItem class, [874](#)
- ragged arrays, [516–517](#)
- RAM. See [random-access memory \(RAM\)](#)
- random access files, [724–730](#)
  - sequential files access vs., [725](#)
- Random class, [255–257](#)
  - nextDouble method, [256](#)
  - nextFloat method, [256](#)
  - nextInt method, [256](#)
  - nextLong method, [256](#)

- random numbers, [255–257](#)
- random-access memory (RAM), [4](#), [11](#)
- RandomAccessFile class, [725–730](#)
  - constructor, [725](#)
  - files pointer, [727–730](#)
  - readByte method, [727](#)
  - readInt method, [727](#)
  - seek method, [727](#)
  - writeChar method, [726](#)
- ranges, checking, [228–229](#)
- ranking, data type, [69](#)
- raw binary format, [719](#)
- read, priming, [287](#)
- REAL data type (SQL), [1021](#)
- realization relationship, [660](#)
- Rectangle class, constructors, [904–907](#)
- recursion, [985–1006](#)
  - base case, [989](#)
  - binary search algorithm, [999–1002](#)
  - concentric circles, drawing, [994–996](#)

- depth of, [987](#)
- direct, [992](#)
- factorial algorithm, [989–992](#)
- Fibonacci series, [996–997](#)
- greatest common divisor, finding, [998–999](#)
- indirect, [992](#)
- introduction, [985–988](#)
- problem solving with, [988–992](#)
- recursive case, [989](#)
- summing array elements, [992–994](#)
- Towers of Hanoi, [1002–1006](#)
- reference, null, [407–410](#)
- reference copy, [395](#)
- reference variables, [76](#), [137](#)
  - arrays, [464–465](#)
  - this, [410–412](#)
  - uninitialized, [159](#)
- referential integrity, [1070](#)
- Reflection class, [944](#), [953](#)
- regular expression, [581](#)

- relational data, [1069–1084](#)
  - joining data from multiple tables, [1071–1072](#)
  - order entry system, [1073–1084](#)
- relational operators, [195–197](#), [1034](#)
  - $\geq$  (greater than or equal to), [195](#), [196](#), [1034](#)
  - $>$  (greater than), [195](#), [196](#), [1034](#)
  - $\leq$  (less than or equal to), [195](#), [196](#), [1034](#)
  - $<$  (less than), [195](#), [196](#), [1034](#)
  - $\neq$  (not equal to), [196](#), [197](#)
  - $\equiv$  (equal to), [195](#), [196](#), [197](#), [1034](#)
- relationship
  - “has a,” [402](#)
  - “is a,” [602](#), [645](#)
  - realization, [658–659](#)
- reserved words. See [key words](#)
- responsibilities, identifying, [180–182](#)
- result sets
  - meta data, [1063–1069](#)
  - scrollable, [1061–1063](#)
- ResultSet class, [1024–1027](#)

- `absolute()` method, [1062](#)
  - and column content retrieval, [1026–1027](#)
  - and row content retrieval, [1025–1026](#)
  - `first()` method, [1062](#)
  - `getDouble` method, [1027](#)
  - `getInt` method, [1027](#)
  - `getMetaData` method, [1063](#)
  - `getRow` method, [1063](#)
  - `getString` method, [1027](#)
  - `last()` method, [1062](#)
  - `next` method, [1026](#)
  - `next()` method, [1062](#)
  - `previous()` method, [1062](#)
  - `relative()` method, [1062](#)
  - `TYPE_CONCUR_READ_ONLY` constant, [1062](#)
  - `TYPE_CONCUR_UPDATEABLE` constant, [1062](#)
  - `TYPE_FORWARD_ONLY` constant, [1061](#)
  - `TYPE_SCROLL_INSENSITIVE` constant, [1061](#)
  - `TYPE_SCROLL_SENSITIVE` constant, [1062](#)
- `ResultSetMetaData` class, [1063](#)

- `getColumnCount` method, [1063](#)
- `getColumnDisplaySize` method, [1063](#)
- `getColumnName` method, [1063](#)
- `getColumnTypeName` method, [1063](#)
- `getTableName` method, [1063](#)
- `return` statement, [140](#)
- reusability, component, [20](#)
- RGB color system, [818–820](#)
- right to left associativity, [61–62](#)
- root selector, [816–817](#)
- rollback, transaction-processing code, [1085](#)
- `RotateTransition` class, [930–934](#)
  - constructors, [930](#)
  - `setFromAngle` method, [930–931](#)
  - `setToAngle` method, [930–931](#)
- rows, [1019–1021](#)
  - content retrieval, `ResultSet` object and, [1025–1026](#)
  - deleting, [1052–1053](#)
  - inserting, [1044–1047](#)
  - updating, [1047–1052](#)

- Ruby programming language, [7](#)
- running a program, [14](#)
- running total, [303–305](#)
- RuntimeException class, [689](#), [709](#), [712](#)

# S

- SalesCommission class, [248–254](#)
- SalesData class, [475–478](#)
- ScaleTransition class, [935–937](#)
  - constructors, [935](#)
- Scanner class, [87–95](#), [332](#)
  - close method, [325](#), [332–333](#)
  - hasNext method, [328–329](#), [333](#)
  - nextByte method, [89](#), [330](#)
  - nextDouble method, [89](#), [330](#)
  - nextFloat method, [89](#), [330](#)
  - nextInt method, [90](#), [330](#)
  - nextLine method, [90](#), [92–95](#), [333](#)
  - nextLong method, [90](#), [330](#)
  - nextShort method, [90](#), [330](#)
  - reading from a file, [325–333](#)
- Scene class, creating object of, [753](#)
- scene graph, [749](#), [751](#)
  - nodes in (see [nodes](#), [JavaFX](#))

- scenes, JavaFX, [749](#)
  - creating, [751–758](#)
  - size of, [756](#)
- scientific notation, [51–52](#)
- scope, [80–81](#), [172–173](#), [235–236](#)
- screen coordinate system, [895–896](#)
- scrollable result sets, [1061–1062](#)
- search algorithm:
  - binary, [501–504](#)
  - sequential, [494–497](#)
- secondary storage, [2](#), [4–5](#)
- seek( )method, [727](#)
- selection sort algorithm, [497–500](#)
- SELECT statement, [1023–1043](#)
  - string comparisons in, [1038](#)
  - WHERE clause, [1034–1040](#)
- self-documenting programs, [45](#)
- separator bar, [874](#)
- SepiaTone class, [944](#), [951–952](#)
- semicolon, [10–11](#), [32–33](#), [200](#)

- sentinel value, [305–307](#)
- setByX method, [928](#)
- setByY method, [928](#)
- setFromAngle method, [930–931](#)
- setFromX method, [927](#)
- setFromY method, [927](#)
- setId method, [822](#)
- setImage method, [762](#)
- setInterpolator method, [942–943](#)
- setOnMouseClicked method, [972](#)
- setOnMouseDragged method, [972](#)
- setOnMouseEntered method, [972](#)
- setOnMouseExited method, [972](#)
- setOnMouseMoved method, [973](#)
- setOnMousePressed method, [973](#)
- setOnMouseReleased method, [973](#)
- setPadding method
  - HBox class, [766](#)
  - VBox class, [769](#)
- setPrefSize method, [840](#)

- `setPreserveRatio` method, [761–762](#)
- `setRotate` method, [922–924](#)
- `setSelected` method, [825–830](#)
- `setStroke` method, [900](#)
- `setStyle` method, [823](#)
- `setToAngle` method, [930–931](#)
- `setToX` method, [927–928](#)
- `setToY` method, [927–928](#)
- sequence structure, [194](#)
- sequential access, files, [724](#)
  - random file access vs., [725](#)
- sequential search algorithm, [494–497](#)
- `Serializable` interface, [730](#)
- serialization, object, [730–734](#)
- `ServiceQuote` class, [182](#)
- `setActionCommand` method, [774](#)
- `setLength` method, [135–139](#)
- `setWidth` method, [139–140](#)
- shadowing, [172–173](#)
- `Shape` class, [896–897](#)

- `setStroke` method, [900](#)
- shapes, drawing, [895–925](#)
  - `Arc` class, [910–914](#)
  - `Circle` class, [900–903](#)
  - `Ellipse` class, [907–910](#)
  - `Polygon` class, [914–917](#)
  - `Polyline` class, [917–919](#)
  - `Rectangle` class, [904–907](#)
  - screen coordinate system, [895–896](#)
  - stroke color, changing, [900](#)
  - `Text` class, [919–921](#)
- `Short` class, [583](#)
  - `MAX_VALUE` constant, [585](#)
  - `MIN_VALUE` constant, [585](#)
  - `parseShort` method, [584](#)
  - `toString` method, [584](#)
- `showOpenDialog` method, [884](#)
- `showSaveDialog` method, [884](#)
- signature, method, [623](#)
- single-line comments, [81–82](#)

- Slider class
  - methods of, [867](#)
- Slider controls, [866–871](#)
- software, [5](#)
  - application, [5](#)
  - engineering, [23](#)
  - operating system, [5](#)
- solid-state drives, [4–5](#)
- sorting algorithm, [497](#)
- source code, [12](#)
- source file, [12](#)
- spacing, HBox layout container, [765–766](#)
- split method, [581–582](#)
- SQL (Structured Query Language), [1015](#)
  - AVG function, [1040–1041](#)
  - COUNT function, [1041](#)
  - CREATE TABLE statement, [1056–1059](#)
  - data types, [1021–1022](#), [1027](#)
  - DELETE statement, [1052–1053](#)
  - DROP TABLE statement, [1059](#)

- [INSERT statement](#), [1044–1047](#)
- [LIKE operator](#), [1038–1039](#)
- mathematical functions, [1040–1041](#)
- [MAX function](#), [1041](#)
- [MIN function](#), [1041](#)
- [ORDER BY clause](#), [1040](#)
- relational operators, [1034](#)
- [SELECT statement](#), [1023–1043](#)
- statement, passing to DBMS, [1024–1034](#)
- stored procedures, [1086](#)
- string comparisons in, [1038](#)
- [SUM function](#), [1041](#)
- syntax, [1035](#)
- [UPDATE statement](#), [1047–1052](#)
- [WHERE clause](#), [1034–1040](#)
- square root, getting, [66](#)
- stack trace, [705–706](#)
- stages, JavaFX, [749](#), [751](#), [754–756](#)
- stale data, [147](#)
- standard input device, [87](#)

- standard output device, [35](#)
- Statement class, [1024](#), [1061](#), [1062](#)
  - executeQuery method, [1024](#)
  - executeUpdate method, [1045](#), [1048–1049](#), [1053](#)
- statements, [10–11](#)
- static class members, [353–359](#)
  - fields, [354–356](#)
  - methods, [354](#), [356–359](#)
- stored procedures, [1086](#)
- string
  - comparing, [230–235](#)
  - comparisons in SQL, [1038](#)
  - concatenation operator, [43](#), [388](#), [561](#)
  - converting to numbers, [99–102](#)
  - literal, [32](#), [74](#)
  - substring, [551](#)
  - tokenizing, [574–582](#)
  - writing and reading, [723–724](#)
- string argument, formatting, [112–116](#)
- String class, [74–79](#), [130](#)

- `charAt` method, [78](#)
- `compareTo` method, [232–234](#)
- `compareToIgnoreCase` method, [234–235](#)
- comparing objects of, [230–235](#)
- `concat` method, [561–562](#)
- conversion to numbers, [99–102](#)
- creating an object of, [76–78](#)
- `endsWith` method, [552](#)
- `equals` method, [230–232](#)
- `equalsIgnoreCase` method, [234–235](#)
- `getChars` method, [559–561](#)
- `indexOf` method, [555–558](#)
- `lastIndexOf` method, [555–558](#)
- `length` method, [77, 78](#)
- `regionMatches` method, [554](#)
- `replace` method, [562](#)
- `split` method, [581–582](#)
- `startsWith` method, [551–552](#)
- `substring` method, [558–559](#)
- `toCharArray` method, [559–561](#)

- `toLowerCase` method, [78](#)
- `toUpperCase` method, [78](#)
- `trim` method, [562](#)
- `valueOf` method, [563](#)
- `StringBuffer` class, [574](#)
- `StringBuilder` class, [565–574](#)
  - `append` methods, [567](#)
  - `charAt` method, [567](#)
  - constructors, [566](#)
  - `delete` method, [569–570](#)
  - `deleteCharAt` method, [569–570](#)
  - `getChars` methods, [567](#)
  - `indexOf` methods, [567](#)
  - `insert` methods, [567–568](#)
  - `lastIndexOf` methods, [567](#)
  - `length` methods, [570](#)
  - `replace` methods, [568–569](#)
  - `setCharAt` method, [569–570](#)
  - `substring` method, [567](#)
  - `toString` methods, [570](#)

- StringTokenizer class, [575–580](#)
  - constructors, [576](#)
  - countTokens method, [577](#)
  - hasMoreTokens method, [577](#)
  - nextTokens method, [577](#)
- String.format method, [102, 114–115](#)
- StrokeTransition class, [938–939](#)
- strongly typed language, [51, 68](#)
- style rules, [810](#)
  - inline, [823](#)
- style sheet, [809](#)
- subclass, [23, 602](#)
- submenu, [874](#)
- subscript, [447–448](#)
- substring, [551–564](#)
- SUM function (SQL), [1041](#)
- Sun Microsystems, [8](#)
- super key word, [612–619, 623](#)
- superclass, [23, 602](#)
  - constructor, [610–619](#)

- method overriding, [620–628](#)
- switch statement, [238–246](#)
  - break key word, [239–243](#)
  - case key word, [239–245](#)
  - enumerated types, with, [423–424](#)
- syntax, [9](#)
- syntax errors, [12](#), [14](#)
- System class, [36](#)
  - exit method, [98](#)
  - in object, [88](#)
  - out object, [36](#)
- System.out.print method. See [print method](#)
- System.out.printf method. See [printf method](#)
- System.out.println method. See [println method](#)

# T

- tables, [1019–1022](#)
  - creating, [1056–1059](#)
  - deleting, [1059](#)
  - multiple, joining data from, [1059–1060](#)
- tabs, in output, [39–40](#)
- telephone numbers, formatting, [570–573](#)
- ternary operator, [58](#)
- TestScoreReader class, [587–590](#)
- Text class, [919–921](#)
- text editor, [12](#)
- TextArea controls, [871–873](#)
- TextField control, [748, 749](#)
  - reading input with, [785–789](#)
- this reference variable, [410–412](#)
- thread, [98](#)
- throw statement, [710–711](#)
- Throwable class, [689](#)
- throws clause, [320, 709–710, 711](#)

- time sharing, [5](#)
- tokenizing strings, [574–583](#)
- tokens, defined, [575](#)
- `toString` method, [388–391](#), [641–642](#)
- Towers of Hanoi, [1002–1006](#)
- trailing else, [218](#)
- transactions, [1084–1086](#)
- transition classes, [926–927
  - `FadeTransition` class, \[940–941\]\(#\)
  - `FillTransition` class, \[939–940\]\(#\)
  - `RotateTransition` class, \[930–934\]\(#\)
  - `StrokeTransition` class, \[938–939\]\(#\)
  - `TranslateTransition` class, \[927–930\]\(#\)](#)
- `TranslateTransition` class, [927–930
  - `setByX` method, \[928\]\(#\)
  - `setByY` method, \[928\]\(#\)
  - `setFromX` method, \[927\]\(#\)
  - `setFromY` method, \[927\]\(#\)
  - `setToX` method, \[927–928\]\(#\)
  - `setToY` method, \[927–928\]\(#\)](#)

- truncation, floating-point, [70](#)
- try statement, [690–693](#)
  - catch block, [690](#)
  - catch clause, [690](#), [706–708](#)
  - finally block, [704](#)
  - finally clause, [703–704](#)
  - try block, [690](#), [696](#)
- two-dimensional arrays
  - arguments, as, [514–516](#)
  - declaration, [505–506](#)
  - displaying the elements in, [512–513](#)
  - initializing, [509–510](#)
  - length field, [510–512](#)
  - ragged arrays, [516–517](#)
  - rows and columns, [505](#)
  - subscripts, [506–507](#)
  - summing all the elements in, [513](#)
  - summing the columns in, [514](#)
  - summing the rows in, [513–514](#)

# U

- UML. See [Unified Modeling Language \(UML\)](#)
- unary operator, [58](#)
- unboxing, [585–586](#)
- unchecked exceptions, [709](#)
- Unicode, [54–55](#)
- Unified Modeling Language (UML), [132–133](#)
  - abstract classes and methods, [653](#)
  - access specification, [147](#), [633](#)
  - aggregation, [404–405](#)
  - class, diagram for, [132–133](#)
  - constructors, [160](#)
  - data type, [147–149](#)
  - inheritance, [609](#), [639](#)
  - interfaces in, [660](#)
  - parameters, [147–149](#)
- uniform resource locator, [956](#)
- uninitialized reference variables, [159](#)
- Universal Serial Bus (USB) drives, [5](#)

- UNIX, [5](#), [12](#)
- UPDATE statement, [1047–1052](#)
- user-controlled loop, [292](#), [297](#)
- USB drives. See [Universal Serial Bus \(USB\) drives](#)
- UTF-8 encoding, [723](#)

# V

- VARCHAR data type (SQL), [1021](#)
- variable-length argument lists, [519–522](#)
- variable(s), [11–12](#), [41–46](#)
  - assignment, [55–56](#)
  - class-type, [75–76](#)
  - declarations, [42](#)
  - defined, [11](#)
  - initialization, [55–56](#)
  - local, [80](#), [172](#)
  - parameter, [135–136](#), [138–139](#), [147–151](#), [172](#)
  - primitive-type, [75](#)
  - reference, [76](#)
  - scope, [80–81](#), [172–173](#), [235–236](#)
  - shadowing, [172–173](#)
  - value hold one at a time by, [56–57](#)
- VBox layout container, [768–770](#)
- vertex, [914](#)
- Visual Basic programming language, [7](#)

# W

- WHERE clause, [1034–1040](#), [1048](#), [1053](#)
- while loop
  - input validation, used for, [286–289](#)
  - programming style, [283–284](#)
- widening conversion, [69–71](#)
- wildcard import statements, [173–174](#)
- Windows operating system, [5](#), [12](#)
- wrapper classes, [543–544](#)
  - Byte, [583](#)
  - Character, [544–551](#)
  - Double, [583](#)
  - Float, [583](#)
  - Integer, [583](#)
  - Long, [583](#)
  - Short, [583](#)

# Credits

Cover image © Dimitris66/E+/Getty Images

Figure 1.2a © iko/Shutterstock

Figure 1.2b © Nikita Rogul/Shutterstock

Figure 1.2c © Feng Yu/Shutterstock

Figure 1.2d © Chiyacat/Shutterstock

Figure 1.2e © Eikostas/Shutterstock

Figure 1.2f © Tkemot/Shutterstock

Figure 1.2g © Vitaly Korovin/Shutterstock

Figure 1.2h © Lusoimages/Shutterstock

Figure 1.2i © jocic/Shutterstock

Figure 1.2j © Best Pictures here/Shutterstock

Figure 1.2k © Peter Guess/Shutterstock

Figure 1.2l © Aquila/Shutterstock

Figure 1.2m © Andre Nitsievsky/Shutterstock

Figure 01.07 © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 2 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

Chapter 2 Oracle screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 5 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 11 Microsoft screenshots - SEE MICROSOFT AGREEMENT FOR FULL CREDIT LINE.

Chapter 11 JavaFX screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 12 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 13 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 14 Java screenshot © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 15 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Appendix D Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Appendix E Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 16 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 17 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 18 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

Chapter 19 Java screenshots © 1995–2016, Oracle and/or its affiliates. All rights reserved.

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED AS PART OF THE SERVICES FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND.

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THE SERVICES. THE DOCUMENTS AND RELATED GRAPHICS CONTAINED HEREIN COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME. PARTIAL SCREEN SHOTS MAY BE VIEWED IN FULL WITHIN THE SOFTWARE VERSION SPECIFIED.

# Contents

1. [Location of Videonotes in the Text](#)
2. [Starting Out with Java™ Early Objects](#)
3. [Starting Out with Java™ Early Objects](#)
4. [Contents in Brief](#)
5. [Contents](#)
6. [Preface](#)
  1. [Early Objects, Late Graphics](#)
  2. [Changes in the Sixth Edition](#)
  3. [Organization of the Text](#)
  4. [Brief Overview of Each Chapter](#)
  5. [Features of the Text](#)
  6. [Supplements](#)
  7. [Reviewers For Previous Editions](#)
  8. [About the Author](#)
7. [Chapter 1 Introduction to Computers and Java](#)
  1. [Topics](#)
  2. [1.1 Introduction](#)
  3. [1.2 Why Program?](#)
  4. [1.3 Computer Systems: Hardware and Software](#)
    1. [Hardware](#)
      1. [The CPU](#)
      2. [Main Memory](#)
      3. [Secondary Storage](#)
      4. [Input Devices](#)
      5. [Output Devices](#)
    2. [Software](#)
    3. [Checkpoint](#)
  5. [1.4 Programming Languages](#)
    1. [What Is a Program?](#)
    2. [A History of Java](#)
  6. [1.5 What Is a Program Made of?](#)
    1. [Language Elements](#)
      1. [Key Words \(Reserved Words\)](#)

- 2. [Programmer-Defined Names](#)
- 3. [Operators](#)
- 4. [Punctuation](#)
- 2. [Lines and Statements](#)
- 3. [Variables](#)
- 4. [The Compiler and the Java Virtual Machine](#)
  - 1. [Portability](#)
- 5. [Java Software Editions](#)
- 6. [Compiling and Running a Java Program](#)
  - 1. [Integrated Development Environments](#)
    - 1. [Checkpoint](#)
- 7. [1.6 The Programming Process](#)
  - 1. [Checkpoint](#)
- 8. [1.7 Object-Oriented Programming](#)
  - 1. [Component Reusability](#)
  - 2. [An Everyday Example of an Object](#)
  - 3. [Classes and Objects](#)
  - 4. [Inheritance](#)
  - 5. [Software Engineering](#)
    - 1. [Checkpoint](#)
- 9. [Review Questions and Exercises](#)
  - 1. [Multiple Choice](#)
  - 2. [Find the Error](#)
  - 3. [Algorithm Workbench](#)
  - 4. [Predict the Result](#)
  - 5. [Short Answer](#)
- 10. [Programming Challenge](#)
- 8. [Chapter 2 Java Fundamentals](#)
  - 1. [Topics](#)
  - 2. [2.1 The Parts of a Java Program](#)
    - 1. [Checkpoint](#)
  - 3. [2.2 The System.out.print and System.out.println Methods, and the Java API](#)
  - 4. [2.3 Variables and Literals](#)
    - 1. [Displaying Multiple Items with the + Operator](#)
    - 2. [Be Careful with Quotation Marks](#)
    - 3. [More about Literals](#)

4. [Identifiers](#)
5. [Class Names](#)
6. [Checkpoint](#)
5. [\*\*2.4 Primitive Data Types\*\*](#)
  1. [The Integer Data Types](#)
    1. [Integer Literals](#)
  2. [Floating-Point Data Types](#)
    1. [Floating-Point Literals](#)
    2. [Scientific and E Notation](#)
  3. [The boolean Data Type](#)
  4. [The char Data Type](#)
    1. [Unicode](#)
  5. [Variable Assignment and Initialization](#)
  6. [Variables Hold Only One Value at a Time](#)
  7. [Checkpoint](#)
6. [\*\*2.5 Arithmetic Operators\*\*](#)
  1. [Integer Division](#)
  2. [Operator Precedence](#)
  3. [Grouping with Parentheses](#)
  4. [The Math Class](#)
    1. [The Math.pow Method](#)
    2. [The Math.sqrt Method](#)
  5. [Checkpoint](#)
7. [\*\*2.6 Combined Assignment Operators\*\*](#)
  1. [Checkpoint](#)
8. [\*\*2.7 Conversion between Primitive Data Types\*\*](#)
  1. [Mixed Integer Operations](#)
  2. [Other Mixed Mathematical Expressions](#)
    1. [Checkpoint](#)
9. [\*\*2.8 Creating Named Constants with final\*\*](#)
10. [\*\*2.9 The String Class\*\*](#)
  1. [Objects Are Created from Classes](#)
  2. [The String Class](#)
  3. [Primitive-Type Variables and Class-Type Variables](#)
  4. [Creating a String Object](#)
  5. [Checkpoint](#)
11. [\*\*2.10 Scope\*\*](#)

12. [2.11 Comments](#)
  1. [Single-Line Comments](#)
  2. [Multiline Comments](#)
  3. [Documentation Comments](#)
  4. [Checkpoint](#)
13. [2.12 Programming Style](#)
14. [2.13 Reading Keyboard Input](#)
  1. [Reading a Character](#)
  2. [Mixing Calls to nextLine with Calls to Other Scanner Methods](#)
15. [2.14 Dialog Boxes](#)
  1. [Displaying Message Dialogs](#)
  2. [Displaying Input Dialogs](#)
  3. [An Example Program](#)
  4. [Converting String Input to Numbers](#)
  5. [Checkpoint](#)
16. [2.15 Displaying Formatted Output with System.out.printf and String.format](#)
  1. [Format Specifier Syntax](#)
  2. [Precision](#)
  3. [Specifying a Minimum Field Width](#)
    1. [Combining Minimum Field Width and Precision in the Same Format Specifier](#)
  4. [Flags](#)
    1. [Comma Separators](#)
    2. [Padding Numbers with Leading Zeros](#)
    3. [Left-Justified Numbers](#)
  5. [Formatting String Arguments](#)
    1. [The String.format Method](#)
  6. [Checkpoint](#)
17. [2.16 Common Errors to Avoid](#)
18. [Review Questions and Exercises](#)
  1. [Multiple Choice and True/False](#)
  2. [Predict the Output](#)
  3. [Find the Error](#)
  4. [Algorithm Workbench](#)
  5. [Short Answer](#)

19. [Programming Challenges](#)
9. [Chapter 3 A First Look at Classes and Objects](#)
  1. [Topics](#)
  2. [3.1 Classes](#)
    1. [Strings as Objects](#)
    2. [Classes and Instances](#)
    3. [Building a Simple Class Step by Step](#)
      1. [Writing the Code for a Class](#)
      2. [Writing the Code for the Class Attributes](#)
    4. [Writing the setLength Method](#)
      1. [Writing the setwidht Method](#)
      2. [Writing the getLength and getWidth Methods](#)
      3. [Writing the getArea Method](#)
    5. [Accessor and Mutator Methods](#)
    6. [The Importance of Data Hiding](#)
    7. [Avoiding Stale Data](#)
    8. [Showing Access Specification in UML Diagrams](#)
    9. [Data Type and Parameter Notation in UML Diagrams](#)
    10. [Layout of Class Members](#)
    11. [Checkpoint](#)
  3. [3.2 More about Passing Arguments](#)
    1. [Passing Multiple Arguments](#)
    2. [Arguments Are Passed by Value](#)
  4. [3.3 Instance Fields and Methods](#)
    1. [Checkpoint](#)
  5. [3.4 Constructors](#)
    1. [The Default Constructor](#)
    2. [No-Arg Constructors](#)
    3. [Showing Constructors in a UML Diagram](#)
    4. [The String Class Constructor](#)
    5. [Checkpoint](#)
  6. [3.5 A BankAccount Class](#)
  7. [3.6 Classes, Variables, and Scope](#)
    1. [Shadowing](#)
  8. [3.7 Packages and import Statements](#)
    1. [Explicit and Wildcard import Statements](#)
    2. [The java.lang Package](#)

- 3. [Other API Packages](#)
- 9. [3.8 Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities](#)
  - 1. [Finding the Classes](#)
  - 2. [Writing a Description of the Problem Domain](#)
    - 1. [Identify All of the Nouns](#)
    - 2. [Refining the List of Nouns](#)
  - 3. [Identifying a Class's Responsibilities](#)
    - 1. [The Customer class](#)
    - 2. [The car Class](#)
    - 3. [The ServiceQuote Class](#)
  - 4. [This Is Only the Beginning](#)
  - 5. [Checkpoint](#)
- 10. [3.9 Common Errors to Avoid](#)
- 11. [Review Questions and Exercises](#)
  - 1. [Multiple Choice and True/False](#)
  - 2. [Find the Error](#)
  - 3. [Algorithm Workbench](#)
  - 4. [Short Answer](#)
- 12. [Programming Challenges](#)
- 10. [Chapter 4 Decision Structures](#)
  - 1. [Topics](#)
  - 2. [4.1 The if Statement](#)
    - 1. [Using Relational Operators to Form Conditions](#)
    - 2. [Putting It All Together](#)
    - 3. [Programming Style and the if Statement](#)
    - 4. [Be Careful with Semicolons](#)
    - 5. [Having Multiple Conditionally Executed Statements](#)
    - 6. [Flags](#)
    - 7. [Comparing Characters](#)
    - 8. [Checkpoint](#)
  - 3. [4.2 The if-else Statement](#)
    - 1. [Checkpoint](#)
  - 4. [4.3 The Payroll Class](#)
  - 5. [4.4 Nested if Statements](#)
    - 1. [Checkpoint](#)
  - 6. [4.5 The if-else-if Statement](#)

1. [The if-else-if Statement Compared to a Nested Decision Structure](#)
  2. [Checkpoint](#)
  7. [\*\*4.6 Logical Operators\*\*](#)
    1. [The Precedence and Associativity of Logical Operators](#)
    2. [Checking Numeric Ranges with Logical Operators](#)
    3. [Checkpoint](#)
  8. [\*\*4.7 Comparing String Objects\*\*](#)
    1. [Ignoring Case in String Comparisons](#)
    2. [Checkpoint](#)
  9. [\*\*4.8 More about Variable Declaration and Scope\*\*](#)
  10. [\*\*4.9 The Conditional Operator \(Optional\)\*\*](#)
    1. [Checkpoint](#)
  11. [\*\*4.10 The switch Statement\*\*](#)
    1. [Checkpoint](#)
  12. [\*\*4.11 Focus on Problem Solving: The SalesCommission Class\*\*](#)
    1. [Private Methods and Algorithm Decomposition](#)
    2. [The Main Program](#)
  13. [\*\*4.12 Generating Random Numbers with the Random Class\*\*](#)
  14. [\*\*4.13 Common Errors to Avoid\*\*](#)
  15. [\*\*Review Questions and Exercises\*\*](#)
    1. [Multiple Choice and True/False](#)
    2. [Find the Error](#)
    3. [Algorithm Workbench](#)
    4. [Short Answer](#)
  16. [\*\*Programming Challenges\*\*](#)
11. [\*\*Chapter 5 Loops and Files\*\*](#)
    1. [Topics](#)
    2. [\*\*5.1 The Increment and Decrement Operators\*\*](#)
      1. [The Difference between Postfix and Prefix Modes](#)
      2. [Checkpoint](#)
    3. [\*\*5.2 The while Loop\*\*](#)
      1. [The while Loop Is a Pretest Loop](#)
      2. [Infinite Loops](#)
      3. [Don't Forget the Braces with a Block of Statements](#)
      4. [Programming Style and the while Loop](#)
        1. [Checkpoint](#)

4. [5.3 Using the `while` Loop for Input Validation](#)
  1. [Checkpoint](#)
5. [5.4 The `do-while` Loop](#)
6. [5.5 The `for` Loop](#)
  1. [The `for` Loop Is a Pretest Loop](#)
  2. [Avoid Modifying the Control Variable in the Body of the `for` Loop](#)
  3. [Other Forms of the Update Expression](#)
  4. [Declaring a Variable in the `for` Loop's Initialization Expression](#)
  5. [Creating a User Controlled `for` Loop](#)
  6. [Using Multiple Statements in the Initialization and Update Expressions](#)
  7. [Checkpoint](#)
7. [5.6 Running Totals and Sentinel Values](#)
  1. [Using a Sentinel Value](#)
  2. [Checkpoint](#)
8. [5.7 Nested Loops](#)
9. [5.8 The `break` and `continue` Statements](#)
10. [5.9 Deciding Which Loop to Use](#)
11. [5.10 Introduction to File Input and Output](#)
  1. [Using the `PrintWriter` Class to Write Data to a File](#)
    1. [More about the `PrintWriter` Class's `println` Method](#)
    2. [The `PrintWriter` Class's `print` Method](#)
    3. [Adding a `throws` Clause to the Method Header](#)
    4. [An Example Program](#)
    5. [Review](#)
    6. [Appending Data to a File](#)
    7. [Specifying the File Location](#)
    8. [Reading Data from a File](#)
    9. [Reading Lines from a File with the `nextLine` Method](#)
    10. [Adding a `throws` Clause to the Method Header](#)
    11. [Detecting the End of a File](#)
    12. [Reading Primitive Values from a File](#)
    13. [Review](#)
    14. [Checking for a File's Existence](#)
      1. [Checkpoint](#)

12. [5.11 Common Errors to Avoid](#)
13. [Review Questions and Exercises](#)
  1. [Multiple Choice and True/False](#)
  2. [Find the Error](#)
  3. [Algorithm Workbench](#)
  4. [Short Answer](#)

14. [Programming Challenges](#)

12. [Chapter 6 A Second Look at Classes and Objects](#)

1. [Topics](#)
2. [6.1 Static Class Members](#)
  1. [A Quick Review of Instance Fields and Instance Methods](#)
  2. [Static Members](#)
  3. [Static Fields](#)
  4. [Static Methods](#)
  5. [The Math Class](#)
  6. [Checkpoint](#)
3. [6.2 Overloaded Methods](#)
  1. [Method Signatures](#)
4. [6.3 Overloaded Constructors](#)
  1. [The Default Constructor Revisited](#)
  2. [The InventoryItem Class](#)
  3. [Checkpoint](#)
5. [6.4 Passing Objects as Arguments to Methods](#)
  1. [Checkpoint](#)
6. [6.5 Returning Objects from Methods](#)
  1. [Checkpoint](#)
7. [6.6 The `toString` Method](#)
8. [6.7 Writing an `equals` Method](#)
9. [6.8 Methods That Copy Objects](#)
  1. [Copy Constructors](#)
10. [6.9 Aggregation](#)
  1. [Aggregation in UML Diagrams](#)
  2. [Security Issues with Aggregate Classes](#)
    1. [Perform Deep Copies When Creating Field Objects](#)
    2. [Return Copies of Field Objects, Not the Original Objects](#)
    3. [Avoid Using `null` References](#)
    4. [Checkpoint](#)

11. [6.10 The `this` Reference Variable](#)
  1. [Using `this` to Overcome Shadowing](#)
  2. [Using `this` to Call an Overloaded Constructor from Another Constructor](#)
    1. [Checkpoint](#)
12. [6.11 Inner Classes](#)
13. [6.12 Enumerated Types](#)
  1. [Enumerated Types Are Specialized Classes](#)
  2. [Switching on an Enumerated Type](#)
    1. [Checkpoint](#)
14. [6.13 Garbage Collection](#)
15. [6.14 Focus on Object-Oriented Design: Class Collaboration](#)
  1. [Determining Class Collaborations with CRC Cards](#)
16. [6.15 Common Errors to Avoid](#)
17. [Review Questions and Exercises](#)
  1. [Multiple Choice and True/False](#)
  2. [Find the Error](#)
  3. [Algorithm Workbench](#)
  4. [Short Answer](#)
18. [Programming Challenges](#)
13. [Chapter 7 Arrays and the `ArrayList` Class](#)
  1. [Topics](#)
  2. [7.1 Introduction to Arrays](#)
    1. [Accessing Array Elements](#)
    2. [Inputting and Outputting Array Contents](#)
    3. [Java Performs Bounds Checking](#)
    4. [Watch Out for Off-by-One Errors](#)
    5. [Array Initialization](#)
    6. [Alternate Array Declaration Notation](#)
    7. [Checkpoint](#)
  3. [7.2 Processing Array Contents](#)
    1. [Array Length](#)
    2. [The Enhanced `for` Loop](#)
      1. [The Enhanced `for` Loop versus the Traditional `for` Loop](#)
    3. [Letting the User Specify an Array's Size](#)
    4. [Reassigning Array Reference Variables](#)
    5. [Copying Arrays](#)

- 6. [Checkpoint](#)
- 4. [7.3 Passing Arrays as Arguments to Methods](#)
  - 1. [Checkpoint](#)
- 5. [7.4 Some Useful Array Algorithms and Operations](#)
  - 1. [Comparing Arrays](#)
  - 2. [Summing the Values in a Numeric Array](#)
  - 3. [Getting the Average of the Values in a Numeric Array](#)
  - 4. [Finding the Highest and Lowest Values in a Numeric Array](#)
  - 5. [The SalesData Class](#)
  - 6. [Partially Filled Arrays](#)
  - 7. [Working with Arrays and Files](#)
- 6. [7.5 Returning Arrays from Methods](#)
- 7. [7.6 String Arrays](#)
  - 1. [Calling String Methods from an Array Element](#)
  - 2. [Checkpoint](#)
- 8. [7.7 Arrays of Objects](#)
  - 1. [Checkpoint](#)
- 9. [7.8 The Sequential Search Algorithm](#)
- 10. [7.9 The Selection Sort and the Binary Search Algorithms](#)
  - 1. [The Selection Sort Algorithm](#)
  - 2. [The Binary Search Algorithm](#)
  - 3. [Checkpoint](#)
- 11. [7.10 Two-Dimensional Arrays](#)
  - 1. [Initializing a Two-Dimensional Array](#)
  - 2. [The length Field in a Two-Dimensional Array](#)
  - 3. [Displaying All the Elements of a Two-Dimensional Array](#)
  - 4. [Summing All the Elements of a Two-Dimensional Array](#)
  - 5. [Summing the Rows of a Two-Dimensional Array](#)
  - 6. [Summing the Columns of a Two-Dimensional Array](#)
  - 7. [Passing Two-Dimensional Arrays to Methods](#)
  - 8. [Ragged Arrays](#)
- 12. [7.11 Arrays with Three or More Dimensions](#)
  - 1. [Checkpoint](#)
- 13. [7.12 Command-Line Arguments and Variable-Length Argument Lists](#)
  - 1. [Command-Line Arguments](#)
  - 2. [Variable-Length Argument Lists](#)

14. [7.13 The ArrayList Class](#)

1. [Creating and Using an ArrayList Object](#)
2. [Using the Enhanced for Loop with an ArrayList](#)
3. [The ArrayList Class's toString method](#)
4. [Removing an Item from an ArrayList](#)
5. [Inserting an Item](#)
6. [Replacing an Item](#)
7. [Capacity](#)
8. [Using the Diamond Operator for Type Inference \(Java 7\)](#)
9. [Checkpoint](#)

15. [7.14 Common Errors to Avoid](#)

16. [Review Questions and Exercises](#)

1. [Multiple Choice and True/False](#)
2. [Find the Error](#)
3. [Algorithm Workbench](#)
4. [Short Answer](#)

17. [Programming Challenges](#)

14. [Chapter 8 Text Processing and Wrapper Classes](#)

1. [Topics](#)
2. [8.1 Introduction to Wrapper Classes](#)
3. [8.2 Character Testing and Conversion with the Character Class](#)
  1. [Character Case Conversion](#)
  2. [Checkpoint](#)
4. [8.3 More about String Objects](#)
  1. [Searching for Substrings](#)
    1. [The startswith and endswith Methods](#)
    2. [The regionMatches Methods](#)
    3. [Finding Characters with the indexOf and lastIndexOf Methods](#)
    4. [Finding Substrings with the indexOf and lastIndexOf Methods](#)
  2. [Extracting Substrings](#)
    1. [The substring Methods](#)
    2. [The getChars and toCharArray Methods](#)
  3. [Methods That Return a Modified String](#)
  4. [The Static valueOf Methods](#)
  5. [Checkpoint](#)

5. [8.4 The StringBuilder Class](#)
  1. [The StringBuilder Constructors](#)
  2. [Other StringBuilder Methods](#)
    1. [The append Methods](#)
    2. [The insert Methods](#)
    3. [The replace Method](#)
    4. [The delete, deleteCharAt, and setCharAt Methods](#)
    5. [The toString Method](#)
  3. [Checkpoint](#)
6. [8.5 Tokenizing Strings](#)
  1. [The StringTokenizer Class](#)
    1. [Extracting Tokens](#)
    2. [Using Multiple Delimiters](#)
    3. [Trimming a String Before Tokenizing](#)
    4. [The String class's split Method](#)
    5. [Checkpoint](#)
7. [8.6 Wrapper Classes for the Numeric Data Types](#)
  1. [The Parse Methods](#)
  2. [The Static toString Methods](#)
  3. [The toBinaryString, toHexString, and toOctalString Methods](#)
  4. [The MIN\\_VALUE and MAX\\_VALUE Constants](#)
  5. [Autoboxing and Unboxing](#)
  6. [Checkpoint](#)
8. [8.7 Focus on Problem Solving: The TestScoreReader Class](#)
9. [8.8 Common Errors to Avoid](#)
10. [Review Questions and Exercises](#)
  1. [Multiple Choice and True/False](#)
  2. [Find the Error](#)
  3. [Algorithm Workbench](#)
  4. [Short Answer](#)
11. [Programming Challenges](#)
15. [Chapter 9 Inheritance](#)
  1. [Topics](#)
  2. [9.1 What Is Inheritance?](#)
    1. [Generalization and Specialization](#)
    2. [Inheritance and the “Is-a” Relationship](#)
      1. [Fields:](#)

2. [Methods:](#)
3. [Inheritance in UML Diagrams](#)
4. [The Superclass's Constructor](#)
5. [Inheritance Does Not Work in Reverse](#)
6. [Checkpoint](#)
3. [9.2 Calling the Superclass Constructor](#)
  1. [When the Superclass Has No Default or No-Arg Constructor](#)
  2. [Summary of Constructor Issues in Inheritance](#)
  3. [Checkpoint](#)
4. [9.3 Overriding Superclass Methods](#)
  1. [Overloading versus Overriding](#)
  2. [Preventing a Method from Being Overridden](#)
  3. [Checkpoint](#)
5. [9.4 Protected Members](#)
  1. [Package Access](#)
  2. [Checkpoint](#)
6. [9.5 Classes That Inherit from Subclasses](#)
  1. [Class Hierarchies](#)
7. [9.6 The Object Class](#)
  1. [Checkpoint](#)
8. [9.7 Polymorphism](#)
  1. [Polymorphism and Dynamic Binding](#)
  2. [The “Is-a” Relationship Does Not Work in Reverse](#)
  3. [The instanceof Operator](#)
  4. [Checkpoint](#)
9. [9.8 Abstract Classes and Abstract Methods](#)
  1. [Abstract Classes in UML](#)
  2. [Checkpoint](#)
10. [9.9 Interfaces](#)
  1. [An Interface Is a Contract](#)
  2. [Fields in Interfaces](#)
  3. [Implementing Multiple Interfaces](#)
  4. [Interfaces in UML](#)
  5. [Default Methods](#)
  6. [Polymorphism and Interfaces](#)
  7. [Checkpoint](#)
11. [9.10 Anonymous Inner Classes](#)

12. [9.11 Functional Interfaces and Lambda Expressions](#)
  1. [Lambda Expressions That Do Not Return a Value](#)
  2. [Lambda Expressions with Multiple Parameters](#)
  3. [Lambda Expressions with No Parameters](#)
  4. [Explicitly Declaring a Parameter's Data Type](#)
  5. [Using Multiple Statements in the Body of a Lambda Expression](#)
  6. [Accessing Variables Within a Lambda Expression](#)
13. [9.12 Common Errors to Avoid](#)
14. [Review Questions and Exercises](#)
  1. [Multiple Choice and True/False](#)
  2. [Find the Error](#)
  3. [Algorithm Workbench](#)
  4. [Short Answer](#)
15. [Programming Challenges](#)
16. [Chapter 10 Exceptions and Advanced File I/O](#)
  1. [Topics](#)
  2. [10.1 Handling Exceptions](#)
    1. [Exception Classes](#)
    2. [Handling an Exception](#)
    3. [Retrieving the Default Error Message](#)
    4. [Polymorphic References to Exceptions](#)
    5. [Handling Multiple Exceptions](#)
      1. [Using Exception Handlers to Recover from Errors](#)
      2. [Handle Each Exception Only Once in a try Statement](#)
    6. [The finally Clause](#)
    7. [The Stack Trace](#)
    8. [Handling Multiple Exceptions with One catch Clause](#)
    9. [When an Exception Is Not Caught](#)
  10. [Checked and Unchecked Exceptions](#)
    1. [Checkpoint](#)
  3. [10.2 Throwing Exceptions](#)
    1. [Creating Your Own Exception Classes](#)
      1. [Checkpoint](#)
  4. [10.3 Advanced Topics: Binary Files, Random Access Files, and Object Serialization](#)
    1. [Binary Files](#)

1. [Writing Data to a Binary File](#)
  2. [Reading Data from a Binary File](#)
  3. [Writing and Reading Strings](#)
  4. [Appending Data to an Existing Binary File](#)
  2. [Random Access Files](#)
    1. [Reading and Writing with the RandomAccessFile Class](#)
    2. [The File Pointer](#)
  3. [Object Serialization](#)
    4. [Checkpoint](#)
  5. [10.4 Common Errors to Avoid](#)
  6. [Review Questions and Exercises](#)
    1. [Multiple Choice and True/False](#)
    2. [Find the Error](#)
    3. [Algorithm Workbench](#)
    4. [Short Answer](#)
  7. [Programming Challenges](#)
17. [Chapter 11 JavaFX: GUI Programming and Basic Controls](#)
1. [Topics](#)
  2. [11.1 Graphical User Interfaces](#)
    1. [Event-Driven GUI Programs](#)
      1. [Checkpoint](#)
  3. [11.2 Introduction to JavaFX](#)
    1. [Controls](#)
    2. [Stages and Scenes](#)
    3. [The Application Class](#)
      1. [Checkpoint](#)
  4. [11.3 Creating Scenes](#)
    1. [Creating Controls](#)
    2. [Creating Layout Containers](#)
    3. [Creating a Scene Object](#)
    4. [Adding the Scene Object to the Stage](#)
    5. [Setting the Size of the Scene](#)
    6. [Aligning Controls in an HBox Layout Container](#)
      1. [Checkpoint](#)
  5. [11.4 Displaying Images](#)
    1. [Loading Images from an Internet Location](#)
    2. [Setting the Size of an Image](#)

- 3. [Preserving the Image's Aspect Ratio](#)
- 4. [Changing an ImageView's Image](#)
  - 1. [Checkpoint](#)
- 6. [11.5 More about the HBox, VBox, and GridPane Layout Containers](#)
  - 1. [The HBox Layout Container](#)
    - 1. [Spacing](#)
    - 2. [Padding](#)
  - 2. [The VBox Layout Container](#)
  - 3. [The GridPane Layout Container](#)
  - 4. [Using Multiple Layout Containers in the Same Screen](#)
    - 1. [Checkpoint](#)
- 7. [11.6 Button Controls and Events](#)
  - 1. [Handling Events](#)
  - 2. [Writing Event Handlers](#)
  - 3. [Registering an Event Handler](#)
    - 1. [Checkpoint](#)
- 8. [11.7 Reading Input with TextField Controls](#)
  - 1. [Checkpoint](#)
- 9. [11.8 Using Anonymous Inner Classes and Lambda Expressions to Handle Events](#)
  - 1. [Using Anonymous Inner Classes to Create Event Handlers](#)
  - 2. [Using Lambda Expressions to Create Event Handlers](#)
    - 1. [Checkpoint](#)
- 10. [11.9 The BorderPane Layout Container](#)
  - 1. [Checkpoint](#)
- 11. [11.10 The ObservableList Interface](#)
- 12. [11.11 Common Errors to Avoid](#)
- 13. [Review Questions and Exercises](#)
  - 1. [Multiple Choice and True/False](#)
  - 2. [Find the Error](#)
  - 3. [Algorithm Workbench](#)
  - 4. [Short Answer](#)
- 14. [Programming Challenges](#)
- 18. [Chapter 12 JavaFX: Advanced Controls](#)
  - 1. [Topics](#)
  - 2. [12.1 Styling JavaFX Applications with CSS](#)
    - 1. [Type Selector Names](#)

2. [Style Properties](#)
  3. [Applying a Stylesheet to a JavaFX Application](#)
  4. [Applying Styles to the Root Node](#)
  5. [Specifying Multiple Selectors in the Same Style Definition](#)
  6. [Working with Colors](#)
    1. [Named Colors](#)
  7. [Creating a Custom Style Class Name](#)
  8. [ID Selectors](#)
  9. [Inline Style Rules](#)
  10. [Checkpoint](#)
3. [\*\*12.2 RadioButton Controls\*\*](#)
    1. [Determining in Code Whether a RadioButton Is Selected](#)
    2. [Selecting a RadioButton in Code](#)
    3. [Responding to RadioButton Clicks](#)
    4. [Checkpoint](#)
  4. [\*\*12.3 checkBox Controls\*\*](#)
    1. [Determining in Code Whether a CheckBox Is Selected](#)
    2. [Selecting a CheckBox in Code](#)
    3. [Responding to CheckBox Clicks](#)
    4. [Checkpoint](#)
  5. [\*\*12.4 ListView Controls\*\*](#)
    1. [Retrieving the Selected Item](#)
    2. [Retrieving the Index of the Selected Item](#)
    3. [Responding to Item Selection with an Event Handler](#)
    4. [Adding Items versus Setting Items](#)
    5. [Initializing a ListView with an Array or an ArrayList](#)
    6. [Selection Modes](#)
    7. [Retrieving Multiple Selected Items](#)
    8. [Working With the Elements of an ObservableList](#)
    9. [Converting an ObservableList to an Array](#)
    10. [Using Code to Select an Item in a ListView](#)
    11. [ListView Orientation](#)
    12. [Creating ListViews of Objects Other Than String](#)
    13. [Checkpoint](#)
  6. [\*\*12.5 comboBox Controls\*\*](#)
    1. [Retrieving the Selected Item](#)
    2. [Responding to ComboBox Item Selection with an Event](#)

- [Handler](#)
  - 3. [Editable ComboBoxes](#)
  - 4. [Checkpoint](#)
  - 7. [12.6 Slider Controls](#)
    - 1. [Checkpoint](#)
  - 8. [12.7 TextArea Controls](#)
    - 1. [Checkpoint](#)
  - 9. [12.8 Menus](#)
  - 10. [12.9 The FileChooser Class](#)
    - 1. [Displaying a FileChooser Dialog Box](#)
    - 2. [Checkpoint](#)
  - 11. [12.10 Using Console Output to Debug a GUI Application](#)
  - 12. [12.11 Common Errors to Avoid](#)
  - 13. [Review Questions](#)
    - 1. [Multiple Choice and True/False](#)
    - 2. [Find the Error](#)
    - 3. [Algorithm Workbench](#)
    - 4. [Short Answer](#)
  - 14. [Programming Challenges](#)
  - 19. [Chapter 13 JavaFX: Graphics, Effects, and Media](#)
    - 1. [Topics](#)
    - 2. [13.1 Drawing Shapes](#)
      - 1. [The Screen Coordinate System](#)
      - 2. [The Shape Class and Its Subclasses](#)
      - 3. [The Line Class](#)
      - 4. [Changing the Stroke Color](#)
      - 5. [The circle Class](#)
      - 6. [The Rectangle Class](#)
      - 7. [The Ellipse Class](#)
      - 8. [The Arc Class](#)
      - 9. [The Polygon Class](#)
      - 10. [The Polyline Class](#)
      - 11. [The Text Class](#)
      - 12. [Rotating Nodes](#)
      - 13. [Scaling Nodes](#)
      - 14. [Checkpoint](#)
    - 3. [13.2 Animation](#)

1. [The TranslateTransition Class](#)
2. [The RotateTransition Class](#)
3. [The ScaleTransition Class](#)
4. [The StrokeTransition Class](#)
5. [The FillTransition Class](#)
6. [The FadeTransition Class](#)
7. [Controlling the Animation](#)
8. [Specifying an Interpolator](#)
9. [Checkpoint](#)
4. [\*\*13.3 Effects\*\*](#)
  1. [The DropShadow Class](#)
  2. [The InnerShadow Class](#)
  3. [The ColorAdjust Class](#)
  4. [The BoxBlur, GaussianBlur, and MotionBlur Classes](#)
    1. [The BoxBlur Class](#)
    2. [The GaussianBlur Class](#)
    3. [The MotionBlur Class](#)
  5. [The SepiaTone Class](#)
  6. [The Glow Class](#)
  7. [The Reflection Class](#)
  8. [Combining Effects](#)
  9. [Checkpoint](#)
5. [\*\*13.4 Playing Sound Files\*\*](#)
  1. [Registering an EndOfMedia Event Handler](#)
  2. [Checkpoint](#)
6. [\*\*13.5 Playing Videos\*\*](#)
  1. [Checkpoint](#)
7. [\*\*13.6 Handling Key Events\*\*](#)
  1. [Using an Anonymous Inner Class to Register a Key Event Handler to the Scene](#)
  2. [Using a Lambda Expression to Register a Key Event Handler to the Scene](#)
  3. [Checkpoint](#)
8. [\*\*13.7 Handling Mouse Events\*\*](#)
9. [\*\*13.8 Common Errors to Avoid\*\*](#)
10. [\*\*Review Questions\*\*](#)
  1. [Multiple Choice and True/False](#)

- 2. [Algorithm Workbench](#)
  - 3. [Short Answer](#)
- 11. [Programming Challenges](#)
- 20. [Chapter 14 Recursion](#)
  - 1. [Topics](#)
  - 2. [14.1 Introduction to Recursion](#)
  - 3. [14.2 Solving Problems with Recursion](#)
    - 1. [Direct and Indirect Recursion](#)
    - 2. [Checkpoint](#)
  - 4. [14.3 Examples of Recursive Methods](#)
    - 1. [Summing a Range of Array Elements with Recursion](#)
    - 2. [Drawing Concentric Circles](#)
    - 3. [The Fibonacci Series](#)
    - 4. [Finding the Greatest Common Divisor](#)
  - 5. [14.4 A Recursive Binary Search Method](#)
  - 6. [14.5 The Towers of Hanoi](#)
  - 7. [14.6 Common Errors to Avoid](#)
  - 8. [Review Questions and Exercises](#)
    - 1. [Multiple Choice and True/False](#)
    - 2. [Find the Error](#)
    - 3. [Algorithm Workbench](#)
    - 4. [Short Answer](#)
  - 9. [Programming Challenges](#)
- 21. [Chapter 15 Databases](#)
  - 1. [Topics](#)
  - 2. [15.1 Introduction to Database Management Systems](#)
    - 1. [JDBC](#)
    - 2. [SQL](#)
    - 3. [Using a DBMS](#)
    - 4. [Java DB](#)
    - 5. [Creating the CoffeedB Database](#)
    - 6. [Connecting to the CoffeedB Database](#)
    - 7. [Connecting to a Password-Protected Database](#)
    - 8. [Checkpoint](#)
  - 3. [15.2 Tables, Rows, and Columns](#)
    - 1. [Column Data Types](#)
    - 2. [Primary Keys](#)

- 3. [Checkpoint](#)
- 4. [15.3 Introduction to the SQL SELECT Statement](#)
  - 1. [Passing an SQL Statement to the DBMS](#)
    - 1. [Getting a Row from the ResultSet Object](#)
    - 2. [Getting Columns in a ResultSet Row](#)
    - 3. [More about the SELECT Statement](#)
  - 2. [Specifying a Search Criteria with the WHERE Clause](#)
    - 1. [String Comparisons in a SELECT Statement](#)
    - 2. [Using the LIKE Operator](#)
    - 3. [Using AND and OR](#)
  - 3. [Sorting the Results of a SELECT Query](#)
  - 4. [Mathematical Functions](#)
  - 5. [Checkpoint](#)
- 5. [15.4 Inserting Rows](#)
  - 1. [Inserting Rows with JDBC](#)
  - 2. [Checkpoint](#)
- 6. [15.5 Updating and Deleting Existing Rows](#)
  - 1. [Updating Rows with JDBC](#)
  - 2. [Deleting Rows with the DELETE Statement](#)
  - 3. [Deleting Rows with JDBC](#)
  - 4. [Checkpoint](#)
- 7. [15.6 Creating and Deleting Tables](#)
  - 1. [Removing a Table with the DROP TABLE Statement](#)
  - 2. [Checkpoint](#)
- 8. [15.7 Creating a New Database with JDBC](#)
- 9. [15.8 Scrollable Result Sets](#)
- 10. [15.9 Result Set Metadata](#)
- 11. [15.10 Relational Data](#)
  - 1. [Joining Data from Multiple Tables](#)
  - 2. [An Order Entry System](#)
- 12. [15.11 Advanced Topics](#)
  - 1. [Transactions](#)
  - 2. [Stored Procedures](#)
- 13. [15.12 Common Errors to Avoid](#)
- 14. [Review Questions and Exercises](#)
  - 1. [Multiple Choice and True/False](#)
  - 2. [Find the Error](#)

3. [Algorithm Workbench](#)
4. [Short Answer](#)
15. [Programming Challenges](#)
22. [Appendix A The ASCII/Unicode Characters](#)
23. [Appendix B Operator Precedence and Associativity](#)
24. [Index](#)
  1. [Symbols](#)
  2. [A](#)
  3. [B](#)
  4. [C](#)
  5. [D](#)
  6. [E](#)
  7. [F](#)
  8. [G](#)
  9. [H](#)
  10. [I](#)
  11. [J](#)
  12. [K](#)
  13. [L](#)
  14. [M](#)
  15. [N](#)
  16. [O](#)
  17. [P](#)
  18. [Q](#)
  19. [R](#)
  20. [S](#)
  21. [T](#)
  22. [U](#)
  23. [V](#)
  24. [W](#)
25. [Credits](#)

## List of Illustrations

1. [Figure P-1 Chapter dependencies](#)
2. [Figure 1-1 The organization of a computer system](#)

3. [Figure 1-2 The organization of the CPU](#)
4. [Figure 1-3 Memory bytes and their addresses](#)
5. [Figure 1-4 A variable name represents a location in memory](#)
6. [Figure 1-5 Program development process](#)
7. [Figure 1-6 Java byte code may be run on any computer with a JVM](#)
8. [Figure 1-7 The NetBeans IDE](#)
9. [Figure 1-8 Screen produced by the pay-calculating algorithm](#)
10. [Figure 1-9 Data is passed among procedures](#)
11. [Figure 1-10 An object contains data and procedures](#)
12. [Figure 1-11 Code outside the object interacts with the object's methods](#)
13. [Figure 1-12 The housefly and mosquito objects are instances of the Insect class](#)
14. [Figure 1-13 An example of inheritance](#)
15. [Figure 2-1 The main method header](#)
16. [Figure 2-2 A console window](#)
17. [Figure 2-3 Relationship among the System class, the out object, and the print and println methods](#)
18. [Figure 2-4 Characters and how they are stored in memory](#)
19. [Figure 2-5 Precedence illustrated](#)
20. [Figure 2-6 Primitive data type ranking](#)
21. [Figure 2-7 A primitive-type variable holds the data with which it is associated](#)
22. [Figure 2-8 A String class variable can hold the address of a String object](#)
23. [Figure 2-9 The name variable holds the address of a String object](#)
24. [Figure 2-10 Documentation generated by javadoc](#)
25. [Figure 2-11 Indentation](#)
26. [Figure 2-12 The parts of the statement](#)
27. [Figure 2-13 The keyboard variable references a Scanner object](#)
28. [Figure 2-14 A message dialog and an input dialog](#)
29. [Figure 2-15 Message dialog](#)
30. [Figure 2-16 Input dialog](#)
31. [Figure 2-17 Dialog boxes displayed by the NamesDialog program](#)
32. [Figure 2-18 Dialog boxes displayed by PayrollDialog.java](#)
33. [Figure 2-19 Dialog boxes for Checkpoint 2.35](#)
34. [Figure 2-20 The value of the sales variable is displayed in the place of the %f format specifier](#)
35. [Figure 2-21 The format specifiers and their corresponding arguments](#)

36. [Figure 2-22 The format specifiers and their corresponding arguments](#)
37. [Figure 2-23 The number is displayed in a field that is 20 spaces wide](#)
38. [Figure 2-24 Format specifier that pads with leading zeros](#)
39. [Figure 2-25 Output of Code Listing 2-38](#)
40. [Figure 3-1 The cityName variable references a String object](#)
41. [Figure 3-2 A blueprint and houses built from the blueprint](#)
42. [Figure 3-3 Three variables referencing three String objects](#)
43. [Figure 3-4 General layout of a UML diagram for a class](#)
44. [Figure 3-5 UML diagram for the Rectangle class](#)
45. [Figure 3-6 Header for the setLength method](#)
46. [Figure 3-7 The box variable references a Rectangle class object](#)
47. [Figure 3-8 The argument 10.0 is copied into the len parameter variable](#)
48. [Figure 3-9 The state of the box object after the setLength method executes](#)
49. [Figure 3-10 The value returned from getLength is assigned to size](#)
50. [Figure 3-11 State of the box object](#)
51. [Figure 3-12 UML diagram for the Rectangle class](#)
52. [Figure 3-13 UML diagram for the Rectangle class with parameter and data type notation](#)
53. [Figure 3-14 Typical layout of class members](#)
54. [Figure 3-15 UML diagram](#)
55. [Figure 3-16 Multiple arguments passed to the set method](#)
56. [Figure 3-17 The kitchen, bedroom, and den variables reference Rectangle objects](#)
57. [Figure 3-18 States of the objects after data has been stored in them](#)
58. [Figure 3-19 Fill in the boxes for each field](#)
59. [Figure 3-20 UML diagram for the Rectangle class showing the constructor](#)
60. [Figure 3-21 UML diagram for the BankAccount class](#)
61. [Figure 3-22 UML diagram for the CellPhone class](#)
62. [Figure 3-23 UML diagram for the Customer class](#)
63. [Figure 3-24 UML diagram for the Car class](#)
64. [Figure 3-25 UML diagram for the ServiceQuote class](#)
65. [Figure 4-1 Sequence structure](#)
66. [Figure 4-2 Simple decision structure logic](#)
67. [Figure 4-3 Three-action decision structure logic](#)
68. [Figure 4-4 Logic of the if statements](#)

69. [Figure 4-5 Do not prematurely terminate an if statement with a semicolon](#)
70. [Figure 4-6 An if statement missing its braces](#)
71. [Figure 4-7 Logic of the if-else statement](#)
72. [Figure 4-8 UML diagram for the Payroll class](#)
73. [Figure 4-9 Logic of nested if statements](#)
74. [Figure 4-10 Alignment of if and else clauses](#)
75. [Figure 4-11 Nested decision structure to determine a grade](#)
76. [Figure 4-12 UML diagram for the TestGrade class](#)
77. [Figure 4-13 String comparison of “Mary” and “Mark”](#)
78. [Figure 4-14 A multiple alternative decision structure](#)
79. [Figure 4-15 UML diagram for the SalesCommission class](#)
80. [Figure 4-16 UML diagram for the Die class](#)
81. [Figure 4-17 Troubleshooting a bad Wi-Fi connection](#)
82. [Figure 5-1 Logic of a while loop](#)
83. [Figure 5-2 The while Loop](#)
84. [Figure 5-3 Logic of the example while loop](#)
85. [Figure 5-4 Input validation logic](#)
86. [Figure 5-5 Logic of the do-while loop](#)
87. [Figure 5-6 Sequence of events in the for loop](#)
88. [Figure 5-7 Logic of the for loop](#)
89. [Figure 5-8 Sequence of events with the for loop in Code Listing 5-7](#)
90. [Figure 5-9 Logic of the for loop in Code Listing 5-7](#)
91. [Figure 5-10 Logic for calculating a running total](#)
92. [Figure 5-11 File contents displayed in Notepad](#)
93. [Figure 5-12 Contents of file displayed in Notepad](#)
94. [Figure 5-13 Contents of the file displayed in Notepad](#)
95. [Figure 5-14 File with three lines](#)
96. [Figure 5-15 Initial read position](#)
97. [Figure 5-16 Read position after first line is read](#)
98. [Figure 5-17 Logic of reading a file until the end is reached](#)
99. [Figure 5-18 Contents of Numbers.txt in Notepad](#)
100. [Figure 6-1 All instances of the class share the static field](#)
101. [Figure 6-2 UML diagram for the InventoryItem class](#)
102. [Figure 6-3 Passing a reference as an argument](#)
103. [Figure 6-4 Both item and i reference the same object](#)
104. [Figure 6-5 UML diagram for the Dealer class](#)

- |05. [Figure 6-6 UML diagram for the Player class](#)
- |06. [Figure 6-7 The getData method header](#)
- |07. [Figure 6-8 UML diagram for the Stock class](#)
- |08. [Figure 6-9 The if statement tests the contents of the reference variables, not the contents of the objects that the variables reference](#)
- |09. [Figure 6-10 Both variables reference the same object](#)
- |10. [Figure 6-11 UML diagram for the Instructor class](#)
- |11. [Figure 6-12 UML diagram for the TextBook class](#)
- |12. [Figure 6-13 UML diagram for the Course class](#)
- |13. [Figure 6-14 UML diagram showing aggregation](#)
- |14. [Figure 6-15 The workDay variable references the Day.WEDNESDAY object](#)
- |15. [Figure 6-16 The Day enumerated data type and the ordinal positions of its enum constants](#)
- |16. [Figure 6-17 Both item1 and item2 reference the same object](#)
- |17. [Figure 6-18 The object is only referenced by the item2 variable](#)
- |18. [Figure 6-19 The object is no longer referenced](#)
- |19. [Figure 6-20 CRC card](#)
- |20. [Figure 6-21 UML diagram for Programming Challenge 3](#)
- |21. [Figure 7-1 Variable declarations and their memory allocations](#)
- |22. [Figure 7-2 The numbers array](#)
- |23. [Figure 7-3 Subscripts for the numbers array](#)
- |24. [Figure 7-4 Contents of the array after 20 is assigned to numbers\[0\]](#)
- |25. [Figure 7-5 Contents of the array after 30 is assigned to numbers\[3\]](#)
- |26. [Figure 7-6 Contents of the hours array](#)
- |27. [Figure 7-7 Annotated loop](#)
- |28. [Figure 7-8 The contents of the array after the initialization](#)
- |29. [Figure 7-9 The numbers variable references a ten-element array](#)
- |30. [Figure 7-10 The numbers variable references a five-element array](#)
- |31. [Figure 7-11 Both array1 and array2 reference the same array](#)
- |32. [Figure 7-12 An array reference passed as an argument](#)
- |33. [Figure 7-13 UML diagram for the Grader class](#)
- |34. [Figure 7-14 Array reference return type](#)
- |35. [Figure 7-15 The names variable references a String array](#)
- |36. [Figure 7-16 An uninitialized String array](#)
- |37. [Figure 7-17 The inventory variable references an array of references](#)
- |38. [Figure 7-18 Each element of the array references an object](#)

- |39. [Figure 7-19 Values in an array](#)
- |40. [Figure 7-20 Values in array after first swap](#)
- |41. [Figure 7-21 Values in array after second swap](#)
- |42. [Figure 7-22 Values in array after third swap](#)
- |43. [Figure 7-23 Values in array after fourth swap](#)
- |44. [Figure 7-24 Values in array after fifth swap](#)
- |45. [Figure 7-25 Rows and columns](#)
- |46. [Figure 7-26 Declaration of a two-dimensional array](#)
- |47. [Figure 7-27 Subscripts for each element of the scores array](#)
- |48. [Figure 7-28 Division and quarter data stored in the sales array](#)
- |49. [Figure 7-29 The numbers array](#)
- |50. [Figure 7-30 The numbers array is an array of arrays](#)
- |51. [Figure 7-31 A three-dimensional array](#)
- |52. [Figure 7-32 Lo Shu Magic Square](#)
- |53. [Figure 7-33 Row, column, and diagonal sums in the Lo Shu Magic Square](#)
- |54. [Figure 8-1 The fullName and lastName variables reference separate objects](#)
- |55. [Figure 8-2 The String object containing “George” is no longer referenced](#)
- |56. [Figure 8-3 Microsoft Excel spreadsheet](#)
- |57. [Figure 9-1 Bumblebees and grasshoppers are specialized versions of an insect](#)
- |58. [Figure 9-2 UML diagram for the GradedActivity class](#)
- |59. [Figure 9-3 UML diagram for the FinalExam class](#)
- |60. [Figure 9-4 First line of the FinalExam class declaration](#)
- |61. [Figure 9-5 UML diagram showing inheritance](#)
- |62. [Figure 9-6 UML diagram for the Rectangle class](#)
- |63. [Figure 9-7 UML diagram for the Rectangle and Cube classes](#)
- |64. [Figure 9-8 The GradedActivity and CurvedActivity classes](#)
- |65. [Figure 9-9 UML diagram for the GradedActivity2 class](#)
- |66. [Figure 9-10 A chain of inheritance](#)
- |67. [Figure 9-11 The GradedActivity, PassFailActivity, and PassFailExam classes](#)
- |68. [Figure 9-12 Class hierarchy](#)
- |69. [Figure 9-13 The line of inheritance from Object to PassFailExam](#)
- |70. [Figure 9-14 Rectangle and Cube classes](#)

- |71. [Figure 9-15 UML diagram for the Student class](#)
- |72. [Figure 9-16 Realization relationship in a UML diagram](#)
- |73. [Figure 9-21 Creating an instance of an anonymous inner class](#)
- |74. [Figure 10-1 Part of the exception class hierarchy](#)
- |75. [Figure 10-2 Sequence of events with an exception](#)
- |76. [Figure 10-3 Sequence of events with no exception](#)
- |77. [Figure 10-4 Inheritance hierarchy for the NumberFormatException class](#)
- |78. [Figure 10-5 UML diagram for the InventoryItem class](#)
- |79. [Figure 10-6 UML diagram for the BankAccount class](#)
- |80. [Figure 10-7 The number 1297 expressed as characters](#)
- |81. [Figure 10-8 The number 1297 as a binary number, as it is stored in memory](#)
- |82. [Figure 10-9 Sequential access versus random access](#)
- |83. [Figure 10-10 Layout of the Letters.dat file](#)
- |84. [Figure 11-1 A command line interface](#)
- |85. [Figure 11-2 A window in a graphical user interface](#)
- |86. [Figure 11-3 Interaction with a program in a text environment](#)
- |87. [Figure 11-4 A GUI application](#)
- |88. [Figure 11-5 Controls in a GUI](#)
- |89. [Figure 11-6 Nodes in a scene graph](#)
- |90. [Figure 11-7 Window displayed by MyFirstGUI.java](#)
- |91. [Figure 11-8 Examples of layout containers](#)
- |92. [Figure 11-9 Nodes in the scene graph](#)
- |93. [Figure 11-10 Window displayed by the HelloWorld application](#)
- |94. [Figure 11-11 Window displayed by the modified HelloWorld application](#)
- |95. [Figure 11-12 Window displayed by the HelloWorld2 application](#)
- |96. [Figure 11-13 The ImageDemo application](#)
- |97. [Figure 11-14 Scene graph for the ImageDemo program in Code Listing 11-4](#)
- |98. [Figure 11-15 The HBoxImages application](#)
- |99. [Figure 11-16 Scene graph for the HBoxImages program in Code Listing 11-5](#)
- |00. [Figure 11-17 Images in an HBox with 10 pixels spacing](#)
- |01. [Figure 11-18 The HBoxImagesWithPadding application](#)
- |02. [Figure 11-19 The VBoxImagesWithPadding application](#)
- |03. [Figure 11-20 Scene graph for the VBoxImagesWithPadding program in](#)

## Code Listing 11-7

- ?04. [Figure 11-21 Column and row indexes](#)
- ?05. [Figure 11-22 The GridPaneDemo application](#)
- ?06. [Figure 11-23 A GridPane with visible grid lines](#)
- ?07. [Figure 11-24 The GridPanelImages application](#)
- ?08. [Figure 11-25 Scene graph for the GridPanelImages program in Code Listing 11-9](#)
- ?09. [Figure 11-26 A window with nested layout containers](#)
- ?10. [Figure 11-27 The layout](#)
- ?11. [Figure 11-28 A window with a Button control](#)
- ?12. [Figure 11-29 The ButtonDemo application](#)
- ?13. [Figure 11-30 A Button control firing an ActionEvent](#)
- ?14. [Figure 11-31 The EventDemo application](#)
- ?15. [Figure 11-32 Layout of the controls in the Kilometer Converter application](#)
- ?16. [Figure 11-33 Scene graph for the Kilometer Converter application](#)
- ?17. [Figure 11-34 The Kilometer Converter application](#)
- ?18. [Figure 11-35 Typical GUI organization](#)
- ?19. [Figure 11-36 The regions of a BorderPane layout container](#)
- ?20. [Figure 11-37 The BorderPaneDemo1 GUI](#)
- ?21. [Figure 11-38 The tic-tac-toe application](#)
- ?22. [Figure 11-39 Slot machine application](#)
- ?23. [Figure 12-1 Output of CSSDemo1.java](#)
- ?24. [Figure 12-2 Output of CSSDemo2.java](#)
- ?25. [Figure 12-3 Output of CSSDemo3.java](#)
- ?26. [Figure 12-4 Example Label with styles applied](#)
- ?27. [Figure 12-5 Output of CSSDemo9.java](#)
- ?28. [Figure 12-6 RadioButton controls](#)
- ?29. [Figure 12-7 The Metric Converter application](#)
- ?30. [Figure 12-8 The Metric Converter application with conversions displayed](#)
- ?31. [Figure 12-9 The RadioButtonEvent application](#)
- ?32. [Figure 12-10 CheckBox controls](#)
- ?33. [Figure 12-11 The PizzaToppings application](#)
- ?34. [Figure 12-12 The PizzaToppings application running](#)
- ?35. [Figure 12-13 ListView examples](#)
- ?36. [Figure 12-14 The ListViewDemo1.java application](#)

- ?37. [Figure 12-15 The ListViewDemo2.java application](#)
- ?38. [Figure 12-16 The ListViewDemo3.java application](#)
- ?39. [Figure 12-17 Selection modes](#)
- ?40. [Figure 12-18 The ListViewDemo4.java application](#)
- ?41. [Figure 12-19 A horizontal ListView](#)
- ?42. [Figure 12-20 The ListViewDemo5.java application](#)
- ?43. [Figure 12-21 The ListViewDemo6.java application](#)
- ?44. [Figure 12-22 The ComboBox control](#)
- ?45. [Figure 12-23 The ComboBoxDemo1.java application](#)
- ?46. [Figure 12-24 The ComboBoxDemo2.java application](#)
- ?47. [Figure 12-25 Uneditable versus Editable ComboBoxes](#)
- ?48. [Figure 12-26 A horizontal and a vertical Slider](#)
- ?49. [Figure 12-27 The SliderDemo.java application](#)
- ?50. [Figure 12-28 The TextArea control](#)
- ?51. [Figure 12-29 Example menu system](#)
- ?52. [Figure 12-30 The SimpleMenu.java application](#)
- ?53. [Figure 12-31 Menu system in the TextMenu.java program](#)
- ?54. [Figure 12-32 The TextMenu.java application](#)
- ?55. [Figure 12-33 The File menu with the Alt+F mnemonic](#)
- ?56. [Figure 12-34 A file chooser dialog box for opening a file](#)
- ?57. [Figure 12-35 A file chooser dialog box for saving a file](#)
- ?58. [Figure 12-36 Messages displayed to the console during the application's execution](#)
- ?59. [Figure 13-1 Various pixel locations in a 640 by 480 window](#)
- ?60. [Figure 13-2 Output of Triangle.java](#)
- ?61. [Figure 13-3 Coordinates of the triangle's corners](#)
- ?62. [Figure 13-4 Output of BullsEye.java](#)
- ?63. [Figure 13-5 Output of CheckerBoard.java](#)
- ?64. [Figure 13-6 An ellipse's center point, x-radius, and y-radius](#)
- ?65. [Figure 13-7 Output of Orbit.java](#)
- ?66. [Figure 13-8 Arc properties](#)
- ?67. [Figure 13-9 Types of arcs](#)
- ?68. [Figure 13-10 Output of PieChart.java](#)
- ?69. [Figure 13-11 Vertices of the octagon displayed by Octagon.java](#)
- ?70. [Figure 13-12 Output of Octogon.java](#)
- ?71. [Figure 13-13 Output of PolylineDemo.java](#)
- ?72. [Figure 13-14 Output of StopSign.java](#)

- ?73. [Figure 13-15 Output of RotateBox.java](#)
- ?74. [Figure 13-16 Output of RotateStopSign.java](#)
- ?75. [Figure 13-17 Output of ScaleText.java](#)
- ?76. [Figure 13-18 Output of BallDrop.java](#)
- ?77. [Figure 13-19 A line rotating from 0 degrees to 90 degrees](#)
- ?78. [Figure 13-20 A line starting at 90 degrees, rotating by an additional 45 degrees](#)
- ?79. [Figure 13-21 Output of RotateImage.java](#)
- ?80. [Figure 13-22 An animation in which text is scaled to 5 times its original size](#)
- ?81. [Figure 13-23 Fade transition example](#)
- ?82. [Figure 13-24 Output of DropShadowDemo.java](#)
- ?83. [Figure 13-25 Drop shadow with X and Y offsets of 10](#)
- ?84. [Figure 13-26 A rectangle with an inner shadow](#)
- ?85. [Figure 13-27 Comparison of normal text and text with a BoxBlur effect](#)
- ?86. [Figure 13-28 BlurBox width and height set to 10 pixels](#)
- ?87. [Figure 13-29 Comparison of normal text and text with a GaussianBlur effect](#)
- ?88. [Figure 13-30 Comparison of normal text and text with a MotionBlur effect](#)
- ?89. [Figure 13-31 Motion blur applied to an image](#)
- ?90. [Figure 13-32 An image without, and with, the Glow effect](#)
- ?91. [Figure 13-33 The Reflection effect applied to a Text object](#)
- ?92. [Figure 13-34 Applying DropShadow and Reflection effects to a Text object](#)
- ?93. [Figure 13-35 Applying DropShadow, SepiaTone, and Reflection effects to an image](#)
- ?94. [Figure 13-36 Window displayed by SoundPlayer.java](#)
- ?95. [Figure 13-37 The VideoDemo.java application](#)
- ?96. [Figure 13-38 The VideoPlayer.java application](#)
- ?97. [Figure 13-39 The KeyPressedDemo.java application after 5 keys have been pressed](#)
- ?98. [Figure 13-40 The MoveBall.java application](#)
- ?99. [Figure 13-41 The MouseEventDemo.java application](#)
- ?100. [Figure 13-42 The MouseMovedDemo.java application](#)
- ?101. [Figure 13-43 Hollywood star](#)
- ?102. [Figure 14-1 First two calls of the method](#)

- 303. [Figure 14-2 Total of six calls to the message method](#)
- 304. [Figure 14-3 Control returns to the point after the recursive method call](#)
- 305. [Figure 14-4 Recursive calls to the factorial method](#)
- 306. [Figure 14-5 Circles application](#)
- 307. [Figure 14-6 The pegs and discs in the Towers of Hanoi game](#)
- 308. [Figure 14-7 Steps for moving three pegs](#)
- 309. [Figure 15-1 A Java application interacts with a DBMS, which manipulates data](#)
- 310. [Figure 15-2 A Java application uses the JDBC classes to interact with a DBMS](#)
- 311. [Figure 15-3 The Coffee database table](#)
- 312. [Figure 15-4 Description column](#)
- 313. [Figure 15-5 A ResultSet object contains the results of an SQL query](#)
- 314. [Figure 15-6 ResultSet rows and columns](#)
- 315. [Figure 15-7 The cursor is initially positioned before the first row](#)
- 316. [Figure 15-8 The next method moves the cursor forward](#)
- 317. [Figure 15-9 Rows where Price is greater than 12.00](#)
- 318. [Figure 15-10 Results of SQL statement](#)
- 319. [Figure 15-11 Results of SQL statement](#)
- 320. [Figure 15-12 Results of SQL statement](#)
- 321. [Figure 15-13 Results of SQL statement](#)
- 322. [Figure 15-14 Results of SQL statement](#)
- 323. [Figure 15-15 Results of the SQL statement](#)
- 324. [Figure 15-16 The DBViewer application](#)
- 325. [Figure 15-17 Entity relationship diagram](#)
- 326. [Figure 15-18 Customer table displayed in the DBViewer application](#)
- 327. [Figure 15-19 Order data entered](#)
- 328. [Figure 15-20 Order information viewed in DBViewer](#)

## List of Tables

- 1. [Table 1-1 Programming languages](#)
- 2. [Table 1-2 The common elements of a programming language](#)
- 3. [Table 1-3 The Java key words](#)
- 4. [Table 2-1 Special characters](#)
- 5. [Table 2-2 Common escape sequences](#)

6. [Table 2-3 Literals](#)
7. [Table 2-4 Some variable names](#)
8. [Table 2-5 Primitive data types for numeric data](#)
9. [Table 2-6 Floating-point representations](#)
10. [Table 2-7 Arithmetic operators](#)
11. [Table 2-8 Precedence of arithmetic operators \(highest to lowest\)](#)
12. [Table 2-9 Associativity of arithmetic operators](#)
13. [Table 2-10 Some expressions and their values](#)
14. [Table 2-11 More expressions and their values](#)
15. [Table 2-12 Various assignment statements \(assume x = 6 in each statement\)](#)
16. [Table 2-13 Combined assignment operators](#)
17. [Table 2-14 Example uses of cast operators](#)
18. [Table 2-15 A few String class methods](#)
19. [Table 2-16 Block comments](#)
20. [Table 2-17 Some of the Scanner class methods](#)
21. [Table 2-18 Methods for converting strings to numbers](#)
22. [Table 3-1 Summary of the private and public Access Specifiers for Class Members](#)
23. [Table 3-2 A few of the standard Java packages](#)
24. [Table 4-1 Relational operators](#)
25. [Table 4-2 boolean expressions using relational operators](#)
26. [Table 4-3 Other examples of if statements](#)
27. [Table 4-4 Logical operators](#)
28. [Table 4-5 boolean expressions using logical operators](#)
29. [Table 4-6 Truth table for the && operator](#)
30. [Table 4-7 Truth table for the || operator](#)
31. [Table 4-8 Truth table for the ! operator](#)
32. [Table 4-9 Logical operators in order of precedence](#)
33. [Table 4-10 Precedence of all operators discussed so far](#)
34. [Table 4-11 Sales commission rates](#)
35. [Table 4-12 SalesCommission class fields](#)
36. [Table 4-13 SalesCommission class methods](#)
37. [Table 4-14 Some of the Random class's methods](#)
38. [Table 6-1 The Stock class methods](#)
39. [Table 8-1 Some static Character class methods for testing char values](#)
40. [Table 8-2 Some Character class methods for case conversion](#)

41. [Table 8-3 String methods that search for a substring](#)
42. [Table 8-4 String methods for getting a character or substring's location](#)
43. [Table 8-5 String methods for extracting substrings](#)
44. [Table 8-6 Methods that return a modified copy of a String object](#)
45. [Table 8-7 Some of the String class's valueOf methods](#)
46. [Table 8-8 StringBuilder constructors](#)
47. [Table 8-9 Methods that are common to the String and StringBuilder classes](#)
48. [Table 8-10 The StringBuilder class's delete and deleteCharAt methods](#)
49. [Table 8-11 The StringTokenizer constructors](#)
50. [Table 8-12 Some of the StringTokenizer methods](#)
51. [Table 8-13 Wrapper classes for the numeric primitive data types](#)
52. [Table 8-14 The parse methods](#)
53. [Table 8-15 Morse code](#)
54. [Table 9-1 CurvedActivity class fields](#)
55. [Table 9-2 CurvedActivity class methods](#)
56. [Table 9-3 Accessibility from within the class's package](#)
57. [Table 9-4 Accessibility from outside the class's package](#)
58. [Table 10-1 Some of the DataOutputStream methods](#)
59. [Table 10-2 Some of the DataInputStream methods](#)
60. [Table 11-1 Three of the most simple layout containers](#)
61. [Table 11-2 enum constants of the Pos type](#)
62. [Table 11-3 BorderPane constructors](#)
63. [Table 11-4 A few of the ObservableList methods](#)
64. [Table 12-1 Some of the JavaFX nodes and their corresponding CSS type selectors](#)
65. [Table 12-2 A few CSS style properties for the Labeled class \(inherited by the Label and Button classes\)](#)
66. [Table 12-3 A few CSS style properties for the Region class \(inherited by the Label and Button classes\)](#)
67. [Table 12-4 A few of the named colors](#)
68. [Table 12-5 ListView methods to select items](#)
69. [Table 12-6 Some of the ComboBox methods](#)
70. [Table 12-7 Some of the Slider class methods](#)
71. [Table 12-8 Some of the TextArea class methods](#)
72. [Table 12-9 Skateboard products](#)
73. [Table 12-10 Optional preconference workshops](#)

74. [Table 13-1 Some of the Shape subclasses](#)
75. [Table 13-2 The Line class constructors](#)
76. [Table 13-3 The Circle class constructors](#)
77. [Table 13-4 The Rectangle class constructors](#)
78. [Table 13-5 The Ellipse class constructors](#)
79. [Table 13-6 The Arc class constructors](#)
80. [Table 13-7 Arc types](#)
81. [Table 13-8 The Arc class constructors](#)
82. [Table 13-9 The PolyLine class constructors](#)
83. [Table 13-10 The Text class constructors](#)
84. [Table 13-11 Some of the transition classes](#)
85. [Table 13-12 TranslateTransition class constructors](#)
86. [Table 13-13 RotateTransition class constructors](#)
87. [Table 13-14 ScaleTransition class constructors](#)
88. [Table 13-15 StrokeTransition class constructors](#)
89. [Table 13-16 FillTransition class constructors](#)
90. [Table 13-17 FadeTransition class constructors](#)
91. [Table 13-18 A few methods inherited from the Animation class](#)
92. [Table 13-19 Interpolator constants](#)
93. [Table 13-20 Some of the effect classes](#)
94. [Table 13-21 Some of the DropShadow class methods](#)
95. [Table 13-22 Some of the ColorAdjust class methods](#)
96. [Table 13-23 Some of the Reflection class methods](#)
97. [Table 13-24 Some of the MediaPlayer methods](#)
98. [Table 13-25 Mouse Event Types](#)
99. [Table 15-1 The Coffee database table](#)
100. [Table 15-2 A few of the SQL data types](#)
101. [Table 15-3 A few of the ResultSet methods](#)
102. [Table 15-4 SQL relational operators](#)
103. [Table 15-5 Rows inserted into the Customer table](#)
104. [Table 15-6 A few ResultSetMetaData methods](#)

# Landmarks

1. [Contents in Brief](#)
2. [Frontmatter](#)

3. [Start of Content](#)
4. [backmatter](#)
5. [List of Illustrations](#)
6. [List of Tables](#)

1. [i](#)
2. [ii](#)
3. [iii](#)
4. [iv](#)
5. [v](#)
6. [vi](#)
7. [vii](#)
8. [viii](#)
9. [ix](#)
10. [x](#)
11. [xi](#)
12. [xii](#)
13. [xiii](#)
14. [xiv](#)
15. [xv](#)
16. [xvi](#)
17. [xvii](#)
18. [xviii](#)
19. [xix](#)
20. [xx](#)
21. [xxi](#)
22. [xxii](#)
23. [xxiii](#)
24. [xxiv](#)
25. [xxv](#)
26. [xxvi](#)
27. [1](#)
28. [2](#)
29. [3](#)
30. [4](#)
31. [5](#)
32. [6](#)

- 33. [7](#)
- 34. [8](#)
- 35. [9](#)
- 36. [10](#)
- 37. [11](#)
- 38. [12](#)
- 39. [13](#)
- 40. [14](#)
- 41. [15](#)
- 42. [16](#)
- 43. [17](#)
- 44. [18](#)
- 45. [19](#)
- 46. [20](#)
- 47. [21](#)
- 48. [22](#)
- 49. [23](#)
- 50. [24](#)
- 51. [25](#)
- 52. [26](#)
- 53. [27](#)
- 54. [28](#)
- 55. [29](#)
- 56. [30](#)
- 57. [31](#)
- 58. [32](#)
- 59. [33](#)
- 60. [34](#)
- 61. [35](#)
- 62. [36](#)
- 63. [37](#)
- 64. [38](#)
- 65. [39](#)
- 66. [40](#)
- 67. [41](#)
- 68. [42](#)
- 69. [43](#)

- 70. [44](#)
- 71. [45](#)
- 72. [46](#)
- 73. [47](#)
- 74. [48](#)
- 75. [49](#)
- 76. [50](#)
- 77. [51](#)
- 78. [52](#)
- 79. [53](#)
- 80. [54](#)
- 81. [55](#)
- 82. [56](#)
- 83. [57](#)
- 84. [58](#)
- 85. [59](#)
- 86. [60](#)
- 87. [61](#)
- 88. [62](#)
- 89. [63](#)
- 90. [64](#)
- 91. [65](#)
- 92. [66](#)
- 93. [67](#)
- 94. [68](#)
- 95. [69](#)
- 96. [70](#)
- 97. [71](#)
- 98. [72](#)
- 99. [73](#)
- 100. [74](#)
- 101. [75](#)
- 102. [76](#)
- 103. [77](#)
- 104. [78](#)
- 105. [79](#)
- 106. [80](#)

- |07. [81](#)
- |08. [82](#)
- |09. [83](#)
- |10. [84](#)
- |11. [85](#)
- |12. [86](#)
- |13. [87](#)
- |14. [88](#)
- |15. [89](#)
- |16. [90](#)
- |17. [91](#)
- |18. [92](#)
- |19. [93](#)
- |20. [94](#)
- |21. [95](#)
- |22. [96](#)
- |23. [97](#)
- |24. [98](#)
- |25. [99](#)
- |26. [100](#)
- |27. [101](#)
- |28. [102](#)
- |29. [103](#)
- |30. [104](#)
- |31. [105](#)
- |32. [106](#)
- |33. [107](#)
- |34. [108](#)
- |35. [109](#)
- |36. [110](#)
- |37. [111](#)
- |38. [112](#)
- |39. [113](#)
- |40. [114](#)
- |41. [115](#)
- |42. [116](#)
- |43. [117](#)

- |44. [118](#)
- |45. [119](#)
- |46. [120](#)
- |47. [121](#)
- |48. [122](#)
- |49. [123](#)
- |50. [124](#)
- |51. [125](#)
- |52. [126](#)
- |53. [127](#)
- |54. [128](#)
- |55. [129](#)
- |56. [130](#)
- |57. [131](#)
- |58. [132](#)
- |59. [133](#)
- |60. [134](#)
- |61. [135](#)
- |62. [136](#)
- |63. [137](#)
- |64. [138](#)
- |65. [139](#)
- |66. [140](#)
- |67. [141](#)
- |68. [142](#)
- |69. [143](#)
- |70. [144](#)
- |71. [145](#)
- |72. [146](#)
- |73. [147](#)
- |74. [148](#)
- |75. [149](#)
- |76. [150](#)
- |77. [151](#)
- |78. [152](#)
- |79. [153](#)
- |80. [154](#)

- |81. [155](#)
- |82. [156](#)
- |83. [157](#)
- |84. [158](#)
- |85. [159](#)
- |86. [160](#)
- |87. [161](#)
- |88. [162](#)
- |89. [163](#)
- |90. [164](#)
- |91. [165](#)
- |92. [166](#)
- |93. [167](#)
- |94. [168](#)
- |95. [169](#)
- |96. [170](#)
- |97. [171](#)
- |98. [172](#)
- |99. [173](#)
- |00. [174](#)
- |01. [175](#)
- |02. [176](#)
- |03. [177](#)
- |04. [178](#)
- |05. [179](#)
- |06. [180](#)
- |07. [181](#)
- |08. [182](#)
- |09. [183](#)
- |10. [184](#)
- |11. [185](#)
- |12. [186](#)
- |13. [187](#)
- |14. [188](#)
- |15. [189](#)
- |16. [190](#)
- |17. [191](#)

- ?18. [192](#)
- ?19. [193](#)
- ?20. [194](#)
- ?21. [195](#)
- ?22. [196](#)
- ?23. [197](#)
- ?24. [198](#)
- ?25. [199](#)
- ?26. [200](#)
- ?27. [201](#)
- ?28. [202](#)
- ?29. [203](#)
- ?30. [204](#)
- ?31. [205](#)
- ?32. [206](#)
- ?33. [207](#)
- ?34. [208](#)
- ?35. [209](#)
- ?36. [210](#)
- ?37. [211](#)
- ?38. [212](#)
- ?39. [213](#)
- ?40. [214](#)
- ?41. [215](#)
- ?42. [216](#)
- ?43. [217](#)
- ?44. [218](#)
- ?45. [219](#)
- ?46. [220](#)
- ?47. [221](#)
- ?48. [222](#)
- ?49. [223](#)
- ?50. [224](#)
- ?51. [225](#)
- ?52. [226](#)
- ?53. [227](#)
- ?54. [228](#)

- ?55. [229](#)
- ?56. [230](#)
- ?57. [231](#)
- ?58. [232](#)
- ?59. [233](#)
- ?60. [234](#)
- ?61. [235](#)
- ?62. [236](#)
- ?63. [237](#)
- ?64. [238](#)
- ?65. [239](#)
- ?66. [240](#)
- ?67. [241](#)
- ?68. [242](#)
- ?69. [243](#)
- ?70. [244](#)
- ?71. [245](#)
- ?72. [246](#)
- ?73. [247](#)
- ?74. [248](#)
- ?75. [249](#)
- ?76. [250](#)
- ?77. [251](#)
- ?78. [252](#)
- ?79. [253](#)
- ?80. [254](#)
- ?81. [255](#)
- ?82. [256](#)
- ?83. [257](#)
- ?84. [258](#)
- ?85. [259](#)
- ?86. [260](#)
- ?87. [261](#)
- ?88. [262](#)
- ?89. [263](#)
- ?90. [264](#)
- ?91. [265](#)

- 192. [266](#)
- 193. [267](#)
- 194. [268](#)
- 195. [269](#)
- 196. [270](#)
- 197. [271](#)
- 198. [272](#)
- 199. [273](#)
- 300. [274](#)
- 301. [275](#)
- 302. [276](#)
- 303. [277](#)
- 304. [278](#)
- 305. [279](#)
- 306. [280](#)
- 307. [281](#)
- 308. [282](#)
- 309. [283](#)
- 310. [284](#)
- 311. [285](#)
- 312. [286](#)
- 313. [287](#)
- 314. [288](#)
- 315. [289](#)
- 316. [290](#)
- 317. [291](#)
- 318. [292](#)
- 319. [293](#)
- 320. [294](#)
- 321. [295](#)
- 322. [296](#)
- 323. [297](#)
- 324. [298](#)
- 325. [299](#)
- 326. [300](#)
- 327. [301](#)
- 328. [302](#)

- 329. [303](#)
- 330. [304](#)
- 331. [305](#)
- 332. [306](#)
- 333. [307](#)
- 334. [308](#)
- 335. [309](#)
- 336. [310](#)
- 337. [311](#)
- 338. [312](#)
- 339. [313](#)
- 340. [314](#)
- 341. [315](#)
- 342. [316](#)
- 343. [317](#)
- 344. [318](#)
- 345. [319](#)
- 346. [320](#)
- 347. [321](#)
- 348. [322](#)
- 349. [323](#)
- 350. [324](#)
- 351. [325](#)
- 352. [326](#)
- 353. [327](#)
- 354. [328](#)
- 355. [329](#)
- 356. [330](#)
- 357. [331](#)
- 358. [332](#)
- 359. [333](#)
- 360. [334](#)
- 361. [335](#)
- 362. [336](#)
- 363. [337](#)
- 364. [338](#)
- 365. [339](#)

- 366. [340](#)
- 367. [341](#)
- 368. [342](#)
- 369. [343](#)
- 370. [344](#)
- 371. [345](#)
- 372. [346](#)
- 373. [347](#)
- 374. [348](#)
- 375. [349](#)
- 376. [350](#)
- 377. [351](#)
- 378. [352](#)
- 379. [353](#)
- 380. [354](#)
- 381. [355](#)
- 382. [356](#)
- 383. [357](#)
- 384. [358](#)
- 385. [359](#)
- 386. [360](#)
- 387. [361](#)
- 388. [362](#)
- 389. [363](#)
- 390. [364](#)
- 391. [365](#)
- 392. [366](#)
- 393. [367](#)
- 394. [368](#)
- 395. [369](#)
- 396. [370](#)
- 397. [371](#)
- 398. [372](#)
- 399. [373](#)
- 400. [374](#)
- 401. [375](#)
- 402. [376](#)

- |03. [377](#)
- |04. [378](#)
- |05. [379](#)
- |06. [380](#)
- |07. [381](#)
- |08. [382](#)
- |09. [383](#)
- |10. [384](#)
- |11. [385](#)
- |12. [386](#)
- |13. [387](#)
- |14. [388](#)
- |15. [389](#)
- |16. [390](#)
- |17. [391](#)
- |18. [392](#)
- |19. [393](#)
- |20. [394](#)
- |21. [395](#)
- |22. [396](#)
- |23. [397](#)
- |24. [398](#)
- |25. [399](#)
- |26. [400](#)
- |27. [401](#)
- |28. [402](#)
- |29. [403](#)
- |30. [404](#)
- |31. [405](#)
- |32. [406](#)
- |33. [407](#)
- |34. [408](#)
- |35. [409](#)
- |36. [410](#)
- |37. [411](#)
- |38. [412](#)
- |39. [413](#)

- I40. [414](#)
- I41. [415](#)
- I42. [416](#)
- I43. [417](#)
- I44. [418](#)
- I45. [419](#)
- I46. [420](#)
- I47. [421](#)
- I48. [422](#)
- I49. [423](#)
- I50. [424](#)
- I51. [425](#)
- I52. [426](#)
- I53. [427](#)
- I54. [428](#)
- I55. [429](#)
- I56. [430](#)
- I57. [431](#)
- I58. [432](#)
- I59. [433](#)
- I60. [434](#)
- I61. [435](#)
- I62. [436](#)
- I63. [437](#)
- I64. [438](#)
- I65. [439](#)
- I66. [440](#)
- I67. [441](#)
- I68. [442](#)
- I69. [443](#)
- I70. [444](#)
- I71. [445](#)
- I72. [446](#)
- I73. [447](#)
- I74. [448](#)
- I75. [449](#)
- I76. [450](#)

- 177. [451](#)
- 178. [452](#)
- 179. [453](#)
- 180. [454](#)
- 181. [455](#)
- 182. [456](#)
- 183. [457](#)
- 184. [458](#)
- 185. [459](#)
- 186. [460](#)
- 187. [461](#)
- 188. [462](#)
- 189. [463](#)
- 190. [464](#)
- 191. [465](#)
- 192. [466](#)
- 193. [467](#)
- 194. [468](#)
- 195. [469](#)
- 196. [470](#)
- 197. [471](#)
- 198. [472](#)
- 199. [473](#)
- 200. [474](#)
- 201. [475](#)
- 202. [476](#)
- 203. [477](#)
- 204. [478](#)
- 205. [479](#)
- 206. [480](#)
- 207. [481](#)
- 208. [482](#)
- 209. [483](#)
- 210. [484](#)
- 211. [485](#)
- 212. [486](#)
- 213. [487](#)

- ;14. [488](#)
- ;15. [489](#)
- ;16. [490](#)
- ;17. [491](#)
- ;18. [492](#)
- ;19. [493](#)
- ;20. [494](#)
- ;21. [495](#)
- ;22. [496](#)
- ;23. [497](#)
- ;24. [498](#)
- ;25. [499](#)
- ;26. [500](#)
- ;27. [501](#)
- ;28. [502](#)
- ;29. [503](#)
- ;30. [504](#)
- ;31. [505](#)
- ;32. [506](#)
- ;33. [507](#)
- ;34. [508](#)
- ;35. [509](#)
- ;36. [510](#)
- ;37. [511](#)
- ;38. [512](#)
- ;39. [513](#)
- ;40. [514](#)
- ;41. [515](#)
- ;42. [516](#)
- ;43. [517](#)
- ;44. [518](#)
- ;45. [519](#)
- ;46. [520](#)
- ;47. [521](#)
- ;48. [522](#)
- ;49. [523](#)
- ;50. [524](#)

- ;51. [525](#)
- ;52. [526](#)
- ;53. [527](#)
- ;54. [528](#)
- ;55. [529](#)
- ;56. [530](#)
- ;57. [531](#)
- ;58. [532](#)
- ;59. [533](#)
- ;60. [534](#)
- ;61. [535](#)
- ;62. [536](#)
- ;63. [537](#)
- ;64. [538](#)
- ;65. [539](#)
- ;66. [540](#)
- ;67. [541](#)
- ;68. [542](#)
- ;69. [543](#)
- ;70. [544](#)
- ;71. [545](#)
- ;72. [546](#)
- ;73. [547](#)
- ;74. [548](#)
- ;75. [549](#)
- ;76. [550](#)
- ;77. [551](#)
- ;78. [552](#)
- ;79. [553](#)
- ;80. [554](#)
- ;81. [555](#)
- ;82. [556](#)
- ;83. [557](#)
- ;84. [558](#)
- ;85. [559](#)
- ;86. [560](#)
- ;87. [561](#)

- ↳ 88. [562](#)
- ↳ 89. [563](#)
- ↳ 90. [564](#)
- ↳ 91. [565](#)
- ↳ 92. [566](#)
- ↳ 93. [567](#)
- ↳ 94. [568](#)
- ↳ 95. [569](#)
- ↳ 96. [570](#)
- ↳ 97. [571](#)
- ↳ 98. [572](#)
- ↳ 99. [573](#)
- ↳ 100. [574](#)
- ↳ 101. [575](#)
- ↳ 102. [576](#)
- ↳ 103. [577](#)
- ↳ 104. [578](#)
- ↳ 105. [579](#)
- ↳ 106. [580](#)
- ↳ 107. [581](#)
- ↳ 108. [582](#)
- ↳ 109. [583](#)
- ↳ 110. [584](#)
- ↳ 111. [585](#)
- ↳ 112. [586](#)
- ↳ 113. [587](#)
- ↳ 114. [588](#)
- ↳ 115. [589](#)
- ↳ 116. [590](#)
- ↳ 117. [591](#)
- ↳ 118. [592](#)
- ↳ 119. [593](#)
- ↳ 120. [594](#)
- ↳ 121. [595](#)
- ↳ 122. [596](#)
- ↳ 123. [597](#)
- ↳ 124. [598](#)

- ↳ 25. [599](#)
- ↳ 26. [600](#)
- ↳ 27. [601](#)
- ↳ 28. [602](#)
- ↳ 29. [603](#)
- ↳ 30. [604](#)
- ↳ 31. [605](#)
- ↳ 32. [606](#)
- ↳ 33. [607](#)
- ↳ 34. [608](#)
- ↳ 35. [609](#)
- ↳ 36. [610](#)
- ↳ 37. [611](#)
- ↳ 38. [612](#)
- ↳ 39. [613](#)
- ↳ 40. [614](#)
- ↳ 41. [615](#)
- ↳ 42. [616](#)
- ↳ 43. [617](#)
- ↳ 44. [618](#)
- ↳ 45. [619](#)
- ↳ 46. [620](#)
- ↳ 47. [621](#)
- ↳ 48. [622](#)
- ↳ 49. [623](#)
- ↳ 50. [624](#)
- ↳ 51. [625](#)
- ↳ 52. [626](#)
- ↳ 53. [627](#)
- ↳ 54. [628](#)
- ↳ 55. [629](#)
- ↳ 56. [630](#)
- ↳ 57. [631](#)
- ↳ 58. [632](#)
- ↳ 59. [633](#)
- ↳ 60. [634](#)
- ↳ 61. [635](#)

- ↳ 62. [636](#)
- ↳ 63. [637](#)
- ↳ 64. [638](#)
- ↳ 65. [639](#)
- ↳ 66. [640](#)
- ↳ 67. [641](#)
- ↳ 68. [642](#)
- ↳ 69. [643](#)
- ↳ 70. [644](#)
- ↳ 71. [645](#)
- ↳ 72. [646](#)
- ↳ 73. [647](#)
- ↳ 74. [648](#)
- ↳ 75. [649](#)
- ↳ 76. [650](#)
- ↳ 77. [651](#)
- ↳ 78. [652](#)
- ↳ 79. [653](#)
- ↳ 80. [654](#)
- ↳ 81. [655](#)
- ↳ 82. [656](#)
- ↳ 83. [657](#)
- ↳ 84. [658](#)
- ↳ 85. [659](#)
- ↳ 86. [660](#)
- ↳ 87. [661](#)
- ↳ 88. [662](#)
- ↳ 89. [663](#)
- ↳ 90. [664](#)
- ↳ 91. [665](#)
- ↳ 92. [666](#)
- ↳ 93. [667](#)
- ↳ 94. [668](#)
- ↳ 95. [669](#)
- ↳ 96. [670](#)
- ↳ 97. [671](#)
- ↳ 98. [672](#)

- '99. [673](#)
- '00. [674](#)
- '01. [675](#)
- '02. [676](#)
- '03. [677](#)
- '04. [678](#)
- '05. [679](#)
- '06. [680](#)
- '07. [681](#)
- '08. [682](#)
- '09. [683](#)
- '10. [684](#)
- '11. [685](#)
- '12. [686](#)
- '13. [687](#)
- '14. [688](#)
- '15. [689](#)
- '16. [690](#)
- '17. [691](#)
- '18. [692](#)
- '19. [693](#)
- '20. [694](#)
- '21. [695](#)
- '22. [696](#)
- '23. [697](#)
- '24. [698](#)
- '25. [699](#)
- '26. [700](#)
- '27. [701](#)
- '28. [702](#)
- '29. [703](#)
- '30. [704](#)
- '31. [705](#)
- '32. [706](#)
- '33. [707](#)
- '34. [708](#)
- '35. [709](#)

- '36. [710](#)
- '37. [711](#)
- '38. [712](#)
- '39. [713](#)
- '40. [714](#)
- '41. [715](#)
- '42. [716](#)
- '43. [717](#)
- '44. [718](#)
- '45. [719](#)
- '46. [720](#)
- '47. [721](#)
- '48. [722](#)
- '49. [723](#)
- '50. [724](#)
- '51. [725](#)
- '52. [726](#)
- '53. [727](#)
- '54. [728](#)
- '55. [729](#)
- '56. [730](#)
- '57. [731](#)
- '58. [732](#)
- '59. [733](#)
- '60. [734](#)
- '61. [735](#)
- '62. [736](#)
- '63. [737](#)
- '64. [738](#)
- '65. [739](#)
- '66. [740](#)
- '67. [741](#)
- '68. [742](#)
- '69. [743](#)
- '70. [744](#)
- '71. [745](#)
- '72. [746](#)

- '73. [747](#)
- '74. [748](#)
- '75. [749](#)
- '76. [750](#)
- '77. [751](#)
- '78. [752](#)
- '79. [753](#)
- '80. [754](#)
- '81. [755](#)
- '82. [756](#)
- '83. [757](#)
- '84. [758](#)
- '85. [759](#)
- '86. [760](#)
- '87. [761](#)
- '88. [762](#)
- '89. [763](#)
- '90. [764](#)
- '91. [765](#)
- '92. [766](#)
- '93. [767](#)
- '94. [768](#)
- '95. [769](#)
- '96. [770](#)
- '97. [771](#)
- '98. [772](#)
- '99. [773](#)
- 300. [774](#)
- 301. [775](#)
- 302. [776](#)
- 303. [777](#)
- 304. [778](#)
- 305. [779](#)
- 306. [780](#)
- 307. [781](#)
- 308. [782](#)
- 309. [783](#)

- 310. [784](#)
- 311. [785](#)
- 312. [786](#)
- 313. [787](#)
- 314. [788](#)
- 315. [789](#)
- 316. [790](#)
- 317. [791](#)
- 318. [792](#)
- 319. [793](#)
- 320. [794](#)
- 321. [795](#)
- 322. [796](#)
- 323. [797](#)
- 324. [798](#)
- 325. [799](#)
- 326. [800](#)
- 327. [801](#)
- 328. [802](#)
- 329. [803](#)
- 330. [804](#)
- 331. [805](#)
- 332. [806](#)
- 333. [807](#)
- 334. [808](#)
- 335. [809](#)
- 336. [810](#)
- 337. [811](#)
- 338. [812](#)
- 339. [813](#)
- 340. [814](#)
- 341. [815](#)
- 342. [816](#)
- 343. [817](#)
- 344. [818](#)
- 345. [819](#)
- 346. [820](#)

- 347. [821](#)
- 348. [822](#)
- 349. [823](#)
- 350. [824](#)
- 351. [825](#)
- 352. [826](#)
- 353. [827](#)
- 354. [828](#)
- 355. [829](#)
- 356. [830](#)
- 357. [831](#)
- 358. [832](#)
- 359. [833](#)
- 360. [834](#)
- 361. [835](#)
- 362. [836](#)
- 363. [837](#)
- 364. [838](#)
- 365. [839](#)
- 366. [840](#)
- 367. [841](#)
- 368. [842](#)
- 369. [843](#)
- 370. [844](#)
- 371. [845](#)
- 372. [846](#)
- 373. [847](#)
- 374. [848](#)
- 375. [849](#)
- 376. [850](#)
- 377. [851](#)
- 378. [852](#)
- 379. [853](#)
- 380. [854](#)
- 381. [855](#)
- 382. [856](#)
- 383. [857](#)

- 384. [858](#)
- 385. [859](#)
- 386. [860](#)
- 387. [861](#)
- 388. [862](#)
- 389. [863](#)
- 390. [864](#)
- 391. [865](#)
- 392. [866](#)
- 393. [867](#)
- 394. [868](#)
- 395. [869](#)
- 396. [870](#)
- 397. [871](#)
- 398. [872](#)
- 399. [873](#)
- 400. [874](#)
- 401. [875](#)
- 402. [876](#)
- 403. [877](#)
- 404. [878](#)
- 405. [879](#)
- 406. [880](#)
- 407. [881](#)
- 408. [882](#)
- 409. [883](#)
- 410. [884](#)
- 411. [885](#)
- 412. [886](#)
- 413. [887](#)
- 414. [888](#)
- 415. [889](#)
- 416. [890](#)
- 417. [891](#)
- 418. [892](#)
- 419. [893](#)
- 420. [894](#)

- )21. [895](#)
- )22. [896](#)
- )23. [897](#)
- )24. [898](#)
- )25. [899](#)
- )26. [900](#)
- )27. [901](#)
- )28. [902](#)
- )29. [903](#)
- )30. [904](#)
- )31. [905](#)
- )32. [906](#)
- )33. [907](#)
- )34. [908](#)
- )35. [909](#)
- )36. [910](#)
- )37. [911](#)
- )38. [912](#)
- )39. [913](#)
- )40. [914](#)
- )41. [915](#)
- )42. [916](#)
- )43. [917](#)
- )44. [918](#)
- )45. [919](#)
- )46. [920](#)
- )47. [921](#)
- )48. [922](#)
- )49. [923](#)
- )50. [924](#)
- )51. [925](#)
- )52. [926](#)
- )53. [927](#)
- )54. [928](#)
- )55. [929](#)
- )56. [930](#)
- )57. [931](#)

- )58. [932](#)
- )59. [933](#)
- )60. [934](#)
- )61. [935](#)
- )62. [936](#)
- )63. [937](#)
- )64. [938](#)
- )65. [939](#)
- )66. [940](#)
- )67. [941](#)
- )68. [942](#)
- )69. [943](#)
- )70. [944](#)
- )71. [945](#)
- )72. [946](#)
- )73. [947](#)
- )74. [948](#)
- )75. [949](#)
- )76. [950](#)
- )77. [951](#)
- )78. [952](#)
- )79. [953](#)
- )80. [954](#)
- )81. [955](#)
- )82. [956](#)
- )83. [957](#)
- )84. [958](#)
- )85. [959](#)
- )86. [960](#)
- )87. [961](#)
- )88. [962](#)
- )89. [963](#)
- )90. [964](#)
- )91. [965](#)
- )92. [966](#)
- )93. [967](#)
- )94. [968](#)

- )95. [969](#)
- )96. [970](#)
- )97. [971](#)
- )98. [972](#)
- )99. [973](#)
- )00. [974](#)
- )01. [975](#)
- )02. [976](#)
- )03. [977](#)
- )04. [978](#)
- )05. [979](#)
- )06. [980](#)
- )07. [981](#)
- )08. [982](#)
- )09. [983](#)
- )10. [984](#)
- )11. [985](#)
- )12. [986](#)
- )13. [987](#)
- )14. [988](#)
- )15. [989](#)
- )16. [990](#)
- )17. [991](#)
- )18. [992](#)
- )19. [993](#)
- )20. [994](#)
- )21. [995](#)
- )22. [996](#)
- )23. [997](#)
- )24. [998](#)
- )25. [999](#)
- )26. [1000](#)
- )27. [1001](#)
- )28. [1002](#)
- )29. [1003](#)
- )30. [1004](#)
- )31. [1005](#)

- )32. [1006](#)
- )33. [1007](#)
- )34. [1008](#)
- )35. [1009](#)
- )36. [1010](#)
- )37. [1011](#)
- )38. [1012](#)
- )39. [1013](#)
- )40. [1014](#)
- )41. [1015](#)
- )42. [1016](#)
- )43. [1017](#)
- )44. [1018](#)
- )45. [1019](#)
- )46. [1020](#)
- )47. [1021](#)
- )48. [1022](#)
- )49. [1023](#)
- )50. [1024](#)
- )51. [1025](#)
- )52. [1026](#)
- )53. [1027](#)
- )54. [1028](#)
- )55. [1029](#)
- )56. [1030](#)
- )57. [1031](#)
- )58. [1032](#)
- )59. [1033](#)
- )60. [1034](#)
- )61. [1035](#)
- )62. [1036](#)
- )63. [1037](#)
- )64. [1038](#)
- )65. [1039](#)
- )66. [1040](#)
- )67. [1041](#)
- )68. [1042](#)

- )69. [1043](#)
- )70. [1044](#)
- )71. [1045](#)
- )72. [1046](#)
- )73. [1047](#)
- )74. [1048](#)
- )75. [1049](#)
- )76. [1050](#)
- )77. [1051](#)
- )78. [1052](#)
- )79. [1053](#)
- )80. [1054](#)
- )81. [1055](#)
- )82. [1056](#)
- )83. [1057](#)
- )84. [1058](#)
- )85. [1059](#)
- )86. [1060](#)
- )87. [1061](#)
- )88. [1062](#)
- )89. [1063](#)
- )90. [1064](#)
- )91. [1065](#)
- )92. [1066](#)
- )93. [1067](#)
- )94. [1068](#)
- )95. [1069](#)
- )96. [1070](#)
- )97. [1071](#)
- )98. [1072](#)
- )99. [1073](#)
- )00. [1074](#)
- )01. [1075](#)
- )02. [1076](#)
- )03. [1077](#)
- )04. [1078](#)
- )05. [1079](#)

- |06. [1080](#)
- |07. [1081](#)
- |08. [1082](#)
- |09. [1083](#)
- |10. [1084](#)
- |11. [1085](#)
- |12. [1086](#)
- |13. [1087](#)
- |14. [1088](#)
- |15. [1089](#)
- |16. [1090](#)
- |17. [1091](#)
- |18. [1092](#)
- |19. [1093](#)
- |20. [1094](#)
- |21. [1095](#)
- |22. [1096](#)
- |23. [1097](#)
- |24. [1098](#)
- |25. [1099](#)
- |26. [1100](#)
- |27. [1101](#)
- |28. [1102](#)
- |29. [1103](#)
- |30. [1104](#)
- |31. [1105](#)
- |32. [1106](#)
- |33. [1107](#)
- |34. [1108](#)
- |35. [1109](#)
- |36. [1110](#)

Labeled boxes are arranged in four rows and are connected to each other by arrows, as follows:

- Chapter 8: Text Processing and Wrapper Classes, Chapter 9: Inheritance, Chapter 15: Databases and Chapter 14: Recursion are located in the second row and depends on Chapters 1 - 7 (Cover in Order) Java Fundamentals which is located in first row.
- Chapter 10: Exceptions and Advanced File I/O and Chapter 11 : Java: GUI Programming and Basic Controls are located in third row and depends on Chapter 9.

Some examples in Chapter 15 use JavaFX, which is introduced in Chapter 11 and One example in Chapter 14 uses the Java FXCircle class, which is introduced in Chapter 13.

- Chapter 11 depends on Chapter 12: JavaFX: Advanced Controls, which is located in fourth row and Chapter 13: Chapter 13 JavaFX: Graphics, Effects, and Media, which is located in fourth row depends on chapter 11.

At the left are the images of input devices, including a scanner, joystick, webcam, keyboard, mouse, camera, and electronic drawing pad. Arrows point from these to a box at center containing a central processing unit and main memory (RAM). Arrows point from this box to the right to images of output devices, including monitors, a printer, and speakers. Below the box, and connected to it by bidirectional arrows, are images of secondary storage devices, including a hard drive and a flash drive.

In the center there is a box containing arithmetic logic unit and control unit and there is bidirectional arrow between them. At the left, an arrow going into the box is labeled as instruction (input) and arrow extending right from the box is labeled as result (output).

The steps in the flowchart are as follows:

- Step 1: The programmer uses the text editor to create a java source code file.
- Step 2: The programmer runs the complier, which translates the source code file into byte code file.
- Step 3: The java virtual machine reads and executes each byte code instruction.

The Steps in the flowchart are as follows:

- Step 1: The programmer creates a java source file.
- Step 2: The programmer runs the compiler, which translates the source code file into byte code file.
- Step 3: This byte code file can run on JVM for windows, Linux, MAC and UNIX.

The screenshot of the window is titled “Payroll-NetBeansIDE 8.1” and shows a menu bar with menus as follows:

- File
- Edit
- View
- Navigate
- Source
- Refractor
- Run
- Debug
- Profile
- Team
- Tools
- Window
- Help

Below the menu bar is a panel which contains a program for payroll class and at the left of this panel are two vertical panels arranged in a column.

The top panel contains directory of folders and bottom panel is a Navigator panel.

The input for "How many hours did you work?" is shown as 10 and for "How much do you get paid?" is shown as 15. The Gross pay "\$150" is shown as output.

At left is a box with dashed outline containing the text "Insect class." The box is labeled "The Insect class describes the data attributes and methods that a particular type of object may have." Arrows point from this box to a pair of boxes to the right. The one above contains the text "housefly object" and is labeled "The housefly object is an instance of the Insect class. It has the data attributes and methods described by the Insect class." The one below contains the text "mosquito object" and is labeled "The mosquito object is an instance of the Insect class. It has the data attributes and methods described by the Insect class."

The diagram shows the program code as "public static void main (String [] args)" with an arrow labeled "Name of the Method" pointing to main. The text at the bottom reads "The other parts of this line are necessary for a method to be properly defined."

A command prompt window containing command lines as follows:

```
C:\Users\Tony\Programs>javac Simple.java
```

```
C:\Users\Tony\Programs>java Simple Programming is great fun!
```

```
C:\Users\Tony\Programs>
```

The diagram shows the system class on the top with the text shown as "The System class holds the out object as well as other members."

The second in the hierarchy is object class with the text shown as "The out object is a member of the System class."

At the bottom in the hierarchy are print and println with the text as follows:

print: The print method is a member of the out object.

println: The println method is a member of the out object.

The diagram shows three characters A, B and C, each stored in a 1 by 2 grid. The grid for A shows 00 in the first column and 65 in the second column. The grid for B shows 00 in the first column and 66 in the second column. The grid for C shows 00 in the first column and 67 in the second column.

The top line reads "outcome equals 12 plus 6 divided by 3." An arrow from 12 points to 12 in the next line and another arrow from 6 divided by 3 points to 2 in the next line. The next line reads "outcome equals 12 plus 2." An arrow point from this line to the next line which reads, "outcome equals 14" An arrow points from 12 plus 2 in the middle line to 14 in the bottom line.

The data types are arranged vertically in the descending order of their rank highest rank at the top and lowest rank at the bottom as follows:

- double
- float
- long
- int
- short
- byte

The diagram shows a small box containing number 25 and the text reads, "The number variable holds the actual data with which it is associated."

The diagram shows a box containing text “address” labeled as “The name variable can hold the address of a String object.” An arrow from this box is pointing to an empty box labeled “A String object.”

The diagram shows a box containing text “address” labeled as “The name variable can hold the address of a String object.” An arrow from this box is pointing to a box labeled “A String object.” The text inside the box reads, “Joe Mahoney.”

The screenshot shows the documentation generated by javadoc. It shows a narrow pane "All classes" on the left with text "Comment3" and a wider pane on the right. This pane shows menus as package, class, tree, deprecated, index, and help with class selected. The text in this pane includes Class comment3, constructor summary, Method summary et cetera. Three tabs at the bottom are All methods, static methods, and concrete methods.

The program shown is as follows:

```
// This example is much more readable than Compact.java.
public class Readable
{
(Single indentation) public static void main(String[] args)
(Single indentation) {
(double indentation) int shares = 220;
(double indentation) double averagePrice = 14.67;
(double indentation) System.out.println("There were" + s
(double indentation with some extra space) "shares sold at
(double indentation with some extra space) averagePrice +
}
```

The statement is as follows:

```
Scanner keyboard equals new Scanner(System.in);
```

The “scanner keyboard” is labeled as “This declare a variable named keyboard. The variable can reference an object of the Scanner class.”

The equals sign is labeled as “The equals operator assigns the address of the Scanner object to the keyboard variable.” and “new Scanner(System.in)” is labeled as “This creates a Scanner object in memory. The object will read input from System.in.”

The diagram shows a box containing text “address” labeled as “The keyboard variable can hold the address of a Scanner object.” An arrow from this box is pointing to a box labeled “A Scanner object.” The text inside the box reads, “This Scanner object is configured to read input from System.in.”

In the message dialog box “Message” is written in the title bar and it contains an icon of a lowercase i in a circle and text “This is an input dialog.” with button 'Ok' underneath the text.

In Input dialog box “Input” is written in the title bar and it contains an icon of a question mark in a square box and text “This message is a dialog box”, with 'Ok' underneath the text.

The first input dialog is shown with text “What is your name?” and a text box below it with text "Joe."

The second input dialog is shown with text “What is your last name?” and a text box below it with text "Clondike."

The message dialog box is shown displaying a greeting text as “Hello Joe Clondike.”

The first input dialog is shown with text “What is your name?” and a fill in field below it. The user types “Joe Mahoney” in the fill in field then clicks “OK” button.

The second input dialog is shown with text “How many hours did you work this week?” and a fill in field below it. The user types “40” in the fill in field and then clicks “OK” button.

The third input dialog is shown with text “What is your hourly pay rate?” and a fill in field below it. The user types “20” in the fill in field then clicks “OK” button.

The message dialog box is shown displaying text as “Hello Joe Mahoney. Your gross pay is \$800.0 .”

In message dialog box “Message” is written in the title bar and it contains text “Greetings Earthling” with “Ok” button underneath.

Input dialog box contains text “Input” in the title bar and text “Enter a number” with a fill in field below it and “Ok” and “cancel” button underneath.

The diagram shows a statement as “System.out.printf (“The temperatures are %f and %f degrees. \n”, temp1, temp2);” with an arrow labeled “1” pointing from temp1 to the first %f and another arrow labeled “2” pointing from the temp2 to the second %f.

The diagram shows a statement as “System.out.printf (“%f %f %f\\n”, value1, value2, value3);” with an arrow labeled “1” pointing from value1 to first %f from the left, an arrow labeled “2” pointing from the value2 to the second %f from the left, and an arrow labeled “3” pointing from the value3 to the third %f from the left.

The diagram shows a statement as “System.out.printf (“The number is:%08.1f\\n”, number);” An arrow pointing to 0 is labeled as “Pad with leading zeros,” an arrow pointing to 8 is labeled as “Minimum field width of 8” and an arrow pointing to decimal point 1 is labeled as “Round to one decimal place.”

The diagram shows a box containing text "address." The box is labeled as "The cityName variable holds the address of a String object." An arrow from this box is pointing to a box labeled as "A string object" and containing text "Charleston."

The three rows are as follows:

Row1:

A box containing text "address." The box is labeled as "The person variable holds the address of a String object." An arrow from this box is pointing to a box labeled as "A string object" and containing text "Jenny."

Row2:

A box containing text "address." The box is labeled as "The pet variable holds the address of a String object." An arrow from this box is pointing to a box labeled as "A string object" and containing text "Fido."

Row3:

A box containing text "address." The box is labeled as "The favoriteColor variable holds the address of a String object." An arrow from this box is pointing to a box labeled as "A string object" and containing text "Blue."

A rectangular box with three rows labeled as:

- Row 1: Class name goes here.
- Row 2: Attributes are listed here.
- Row 3: Methods are listed here.

A rectangular box with three rows. Text in each row reads as:

- Row 1: Rectangle.

- Row 2:

- length

- width.

- Row 3:

- setLength()

- setWidth()

- getLength()

- getWidth()

- getArea()

The diagram shows a statement as: public void setLength(double len)

The parts of statement are labeled as follows:

- public: Access specifier
- Void: Return type
- setLength: Method name
- double len: Parameter variable declaration

A diagram shows a box labeled “The box variable holds the address of a Rectangle object” and containing text “address.” A arrow from this box is pointing to another box labeled “A Rectangle object” and containing text “length: 0.0; width: 0.0” with each 0.0 enclosed in a box.

A diagram shows a statement as follows:

```
public void setLength(double len)
{
length = len;
}
```

Above this statement there is another statement as “box.setLength(10.0)” and an arrow from 10.0 is pointing to “len” of the first line of the bottom statement.

A diagram shows a box labeled “The box variable holds the address of a Rectangle object” and containing text “address.” An arrow from this box is pointing to another box labeled “A Rectangle object” containing text “length:10.0; width: 0.0” with 0.0 and 10.0 enclosed in a box.

A diagram shows a statement as follows:

```
size equals box.getLength();
```

Below this is a statement as follows:

```
public double getLength()
{
 return length;
}
```

An arrow pointing from the line “return length;” is pointing to the “size” of the first statement.

A diagram shows a box labeled “The box variable holds the address of a Rectangle object” and containing text “address.” An arrow from this box is pointing to another box labeled “A Rectangle object” containing text “length:10.0; width:20.0” with 10.0 and 20.0 enclosed in a box.

A UML diagram consisting of rectangular box divided into three rows with the following data filled in:

- Top row: Rectangle
- Middle row:
  - private length
  - private width
- Bottom row:
  - public setLength()
  - public setWidth()
  - public getLength()
  - public getWidth()
  - public getArea()

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Rectangle
- Middle row :
  - private length : double
  - private width : double
- Bottom row:
  - public setLength(len : double) : void
  - public setWidth(w : double) : void
  - public getLength() : double
  - public getWidth() : double
  - public getArea() : double

A diagram containing a statement at the top as:

```
public class ClassName
```

Below this statement are two rectangular boxes with text “Field declarations” in the first box and “Method definitions” in the second box.

A UML diagram consisting of rectangular box divided into three rows with the following data filled in:

- Top row: Car
- Middle row:
  - private make
  - private yearModel
- Bottom row:
  - public setMake()
  - public setYearModel()
  - public getMake()
  - public getYearModel()

The diagram shows the program code as follows:

```
box.set(10.0, 20.0);
public void set(double len, double w)
{
length = len;
width = w;
}
```

An arrow from 10.0 in the top row points to “len” in the second row and another arrow from 20.0 in the first row points to the “w” in the second row.

The diagram shows three rows as follows:

- Row 1: A box labeled “The kitchen variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 0.0; width: 0.0” with each 0.0 enclosed in a box.
- Row 2: A box labeled “The bedroom variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 0.0; width: 0.0” with each 0.0 enclosed in a box.
- Row 2: A box labeled “The den variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 0.0; width: 0.0” with each 0.0 enclosed in a box.

The diagram shows three rows as follows:

- Row 1: A box labeled “The kitchen variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 10.0; width: 10.0” with 10.0 and 14.0 enclosed in a box.
- Row 2: A box labeled “The bedroom variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 15.0; width: 12.0” with 15.0 and 12.0 enclosed in a box.
- Row 2: A box labeled “The den variable holds the address of a Rectangle object.” and containing text “address.” An arrows from this box is pointing to another box labeled “A Rectangle object” containing text “length: 20.0; width: 30.0” with 20.0 and 30.0 enclosed in a box.

The diagram shows two rows as follows:

- Row 1: A box labeled “r1” and containing text “address.” An arrow from this box is pointing to another box labeled “A Rectangle object” containing text “length:; width.”
- Row 1: A box labeled “r2” and containing text “address.” An arrow from this box is pointing to a box labeled “A Rectangle object” containing text “length:; width.” In each row, “length:” and “breadth:” are followed by a small box.

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Rectangle
- Middle row :
  - private length : double
  - private width : double
- Bottom row:
  - public Rectangle(len : double, w : double)
  - public setLength(len : double) : void
  - public setWidth(w : double) : void
  - public getLength() : double
  - public getWidth() : double
  - public getArea() : double

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: BankAccount
- Middle row :
  - private balance : double
  - private interestRate : double
  - private interest : double
- Bottom row:
  - public BankAccount(startBalance : double, intRate : double)
  - public deposit(amount : double) : void
  - public withdraw(amount : double) : void
  - public addInterest() : void
  - public getBalance() : double
  - public getInterest() : double

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: CellPhone
- Middle row :
  - private manufact : String
  - private model : String
  - private retailPrice : double
- Bottom row:
  - public CellPhone(man : String, mod : String, price : double);
  - public setManufact(man : String) : void
  - public setModel(mod : String) : void
  - public setRetailPrice(price : double) : void
  - public getManufact() : String
  - public getModel() : String
  - public getRetailPrice() : double

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Customer
- Middle row :
  - private name : String
  - private address : String
  - private phone : String
- Bottom row:
  - public Customer()
  - public setName(n : String) : void
  - public setAddress(a : String) : void
  - public setPhone(p : String) : void
  - public getName() : String
  - public getAddress() : String
  - public getPhone() : String"

The diagram shows a rectangular box divided into three rows with data with the following data filled in:

- Top row: Car
- Middle row :
  - private make : String
  - private model : String
  - private year : int
- Bottom row:
  - public Car()
  - public setMake(m : String) : void
  - public setModel(m : String) : void
  - public setYear(y : int) : void
  - public getMake() : String
  - public getModel() : String
  - public getYear() : int

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: ServiceQuote
- Middle row :
  - private partsCharges : double
  - private laborCharges : double
- Bottom row:
  - public ServiceQuote()
  - public setPartsCharges(c : double) : void
  - public setLaborCharges(c : double) : void
  - public getPartsCharges() : double
  - public getLaborCharges() : double
  - public salesTax() : double
  - public totalCharges() : double

The sequence structure shown is as follows:

```
public class SquareArea
{
 public static void main(String[] args)
 {
 double length, width, area;
 Step 1 :length = 10;
 Step 2 :width = 5;
 Step 3:area = length * width;
 Step 4:System.out.print("The area is" plus area);
 }
}
```

The diagram shows an arrow pointing down to a diamond-shaped box containing the text "Is it cold outside?." An arrow labeled "No" extends downward from the bottom of the diamond. An arrow labeled "Yes" extends from the right side of the diamond and points to a box containing the text "Wear a coat."

An arrow points down to a diamond-shaped box containing the text "Is it cold outside?" An arrow labeled "No" extends downward from the bottom of the diamond. An arrow labeled "Yes" extends from the right side of the diamond and points to a box containing the text "Wear a coat," an arrow from the bottom of this box points to a box containing text "Wear a hat," and an arrow from the bottom of this box points to a box containing text "Wear gloves."

An arrow points down to a diamond-shaped box containing the text “average greater than 95.” An arrow labeled “False” extends downward from the bottom of the diamond. An arrow labeled “True” extends from the right side of the diamond and points to a box containing the text "Display 'That's a great score!'"

A diagram shows the statements: if (BooleanExpression) statement;

In the statements above no semicolon is placed at the end of the first statement and a semicolon is placed at the end of the second statement.

The diagram shows the program code as follows:

- if (sales greater than 50000) bonus equals 500.0;
- commissionRate equals 0.12;
- daysOff plus equals 1;

The statement “bonus equals 500.0;” is labeled as “Only this statement is conditionally executed.” and the last two statements are labeled as "These statements are always executed."

An arrow points down to a diamond-shaped box containing the text "number2 equals equals 0." An arrow labeled "False" extends downward from the left of the diamond and point to a box containing text "Display number1 divided by number2." An arrow labeled "True" extends from the right side of the diamond and points to a box containing the text "Display error message."

The diagram shows a rectangular box with three rows with the following data filled in:

- Top row: Payroll
- Middle row :
  - private hoursWorked : double
  - private payRate : double
- Bottom row:
  - public Payroll()
  - public setHoursWorked(hours : double) : void
  - public setPayRate(rate : double) : void
  - public getHoursWorked() : double
  - public getPayRate() : double
  - public getGrossPay() : double

The flowchart shows an arrow pointing to a diamond box containing statement "salary greater than or equal to 50000." An arrow labeled "False" from the left of the diamond leads to a box containing text "Display 'You must earn at least \$50,000 per year to qualify.'"

An arrow labeled "True" from the right of the diamond extends to another diamond containing text "yearsOnJob greater than or equal to 2."

An arrow labeled "False" from the left of the diamond leads to a box containing text "Display 'You must have been on your current job for at least two years to qualify.'"

An arrow labeled "True" from the right of the diamond leads to a box containing text "Display 'You qualify for the loan.'"

The program is as follows:

```
if (salary greater than or equal to 50000)
{
if (yearsOnJob greater than or equal to 2)
{
System.out.println("You qualify for the loan.");
}
else
{
System.out.println("You must have been on your" plus
"current job for at least" plus
"two years to qualify.");
}
}
else
{
System.out.println("You must earn at least" plus
"$50,000 per year to qualify.");
}
```

In the program above the inner if and else are aligned at the same level and outer if and else are aligned at the same level.

An arrow points downward to a diamond-shaped box containing the text "score less than 60". An arrow labeled "False" extends from the left side of the diamond and points down to another diamond-shaped box, beginning a sequence of three such boxes connected by arrows labeled "False" extending from the left sides. The sequence is as follows:

- score less than 70
- score less than 80
- score less than 90

An arrow labeled "False" extends from the left side of this last diamond and points down to a box containing the text "grade is A." An arrow labeled "True" extends from the right side of each of the four diamonds and points to a box with text as follows:

- For score less than 60: "grade is F"
- For score less than 70: "grade is D"
- For score less than 80: "grade is C."
- For score less than 90: "grade is B."

Arrows extend down from each of the print boxes and converge into a single arrow pointing downward.

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: TestGrade
- Middle row :
  - private score : int
- Bottom row:
  - public TestGrade(s: int)
  - public setScore(s : int) : void
  - public getScore() : int
  - public getLetterGrade() : char

```
if (expression_1)
{
statement
statement
etc.
The annotation for above three rows of codes reads, "If expressio
}
else if (expression_2)
{
statement
statement
etc.
The annotation for above three rows of codes reads, "Otherwise, i
}
Insert as many else if clauses as necessary
else
{
statement
statement
etc.
The annotation for above three rows of codes reads, "These statem
}
```

An arrow points down to a diamond-shaped box containing the text “month” which leads to four input boxes by 4 arrows labeled: “1, 2, 3 and default.” The text in the boxes from left to right is labeled as “Display January,” “Display February,” Display March, and “Display Error: Invalid month.”

```
switch (testExpression)
The annotation for testExpression reads, "The testExpression is a
{
case value_1:
statement;
statement;
etc.
break;
The annotation for above four rows of codes reads, "These stateme
case value_2:
statement;
statement;
etc.
break;
The annotation for above four rows of codes reads, "These stateme
Insert as many case sections as necessary.
case value_N:
statement;
statement;
etc.
break;
The annotation for above four rows of codes reads, "These stateme
default:
statement;
statement;
etc.
break;
The annotation for above four rows of codes reads, "These stateme
}
```

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: SalesCommission
- Middle row :
  - private sales : double
  - private rate : double
  - private commission : double
  - private advance : double
  - private pay : double
- Bottom row:
  - public SalesCommission(s : double, a : double)
  - private setRate() : void
  - private calculatePay() : void
  - public getPay() : double
  - public getCommission() : double
  - public getRate() : double
  - public getAdvance() : double
  - public getSales() : double

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Die
- Middle row :
  - private sides : int
  - private value : int
- Bottom row:
  - public Die(numSides : int)
  - public roll() : void
  - public getSides() : int
  - public getValue() : int

The arrows indicate the flow from top to bottom, as follows:

- Box at top: "Reboot the computer and try to connect."
- Diamond-shaped box: "Did that fix the problem?" If no:
- Box: "Reboot the router and try to connect."
- Diamond-shaped box: "Did that fix the problem?" If no:
- Box: "Make sure the cables between the router & modem are plugged in firmly."
- Diamond-shaped box: "Did that fix the problem?" If no:
- Box: "Move the router to a new location and try to connect."
- Diamond-shaped box: "Did that fix the problem?" If no:
- Box: "Get a new router."

The arrows labeled "Yes" extend from each of the diamond-shaped boxes and converge along with an arrow extending from the last box into a single arrow pointing downward.

An arrow points down to a diamond-shaped box containing the text "Boolean Expression." An arrow labeled "False" extends downward from the bottom of the diamond. An arrow labeled true extends from the right side of the diamond and points to a box containing the text "Statement(s)." An arrow extends from this box and points to the arrow at the top of the diamond box.

"A diagram shows a statement:

```
while (number less than or equal to 5)
{
 System.out.println("Hello");
 number plus plus;
}
```

In the above statement the “less than or equal to” is labeled as “Test this Boolean expression,” the statement within the braces is labeled as “If the Boolean expression is true, perform these statements,” and the arrow is pointing to “while” which is labeled as “After executing the body of the loop, start over.”

An arrow points down to a diamond-shaped box containing the text "number less than or equal to 5." An arrow labeled "False" extends downward from the bottom of the diamond. An arrow labeled true extends from the right side of the diamond and points to a box containing the text "print Hello." An arrow extends from this box and points to another box containing the text "print Add 1 to number" and an arrow from this box points to the arrow at the top of the diamond box.

An arrow points down to a box containing the text "Read the first value" an arrow from the bottom of this box points to a diamond box containing the text "Is the value invalid." An arrow labeled No extends downward from the bottom of the diamond. An arrow labeled Yes extends from the right side of the diamond and points to a box containing the text "Display an error message," an arrow points from this box to a box containing the text "Read another value," and arrow and an arrow from this box points at the top of the diamond box.

An arrow points down to a box containing the text "Statement(s)" an arrow from the bottom of this box points to a diamond box containing the text "Boolean Expression." An arrow labeled false extends downward from the bottom of the diamond. An arrow labeled true extends from the right side of the diamond points to the arrow at the top.

The sequence of events is as follows:

for (count equals 1; count less than or equal to 5; count plus plus)

System.out.println("Hello");

In the code above, "count equals 1" is labeled as "Step 1: Perform the initialization expression," "count less than or equal to 5" is labeled as "Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop," the body of loop "System.out.println('Hello');" is labeled as: "Step 3: Execute the body of the loop,"

The "count plus plus" is labeled as "Step 4: Perform the update expression, then go back to Step 2."

An arrow points down to a box containing the text "Assign 1 to count" an arrow from the bottom of this box points to a diamond box containing the text "count less than or equal to 5." An arrow labeled "False" extends downward from the bottom of the diamond. An arrow labeled "True" extends from the right side of the diamond and points to a box containing the text "println statement," an arrow points from this box points to a box containing the text "Increment count" and an arrow from this box points to the arrow at the top of the diamond box.

The sequence of the events is as follows:

for (number equals 1; number less than or equal to 10; number plus plus)

```
{
 System.out.println(number plus "\t\t" plus
 number times number);
}
```

In the code above “number equals 1;” is labeled as “Step 1: Perform the initialization expression,” the text “number less than or equal to 10;” is labeled as “Step 2: Evaluate the test expression. If it is true, go to Step 3. Otherwise, terminate the loop,” the body of loop “System.out.println(number plus "\t\t" plus number times number);” is labeled as “Step 3: Execute the body of the loop,” “number plus plus“ is labeled as “Step 4: Perform the update expression, then go back to Step 2.”

An arrow points down to a box containing the text "Assign 1 to number" an arrow from the bottom of this box points to a diamond box containing the text "number less than or equal to 5." An arrow labeled "False" extends downward from the bottom of the diamond. An arrow labeled "True" extends from the right side of the diamond and points to a box containing the text "println statement," an arrow points from this box to a box containing the text "Increment number" and an arrow from this box points to an arrow at the top of the diamond box.

An arrow points down to a rectangular box containing the text "Set accumulator to 0" an arrow from the bottom of this box points to a diamond box containing the text "Is there another number to read?" An arrow labeled false extends downward from the bottom of the diamond. An arrow labeled true extends from the right side of the diamond and points to a parallelogram containing the text "Read the next number," an arrow points from this box to a rectangular box containing the text "Add the number to the accumulator" and an arrow from this box points to an arrow at the top of the diamond box.

A screenshot of a Notepad dialog box with text “StudentData.txt – Notepad” written in the title bar and containing six lines of text, as follows:

Jim

95

Karen

98

Bob

82

A screenshot of a Notepad dialog box with text “PersonalData.txt - Notepad” written in the title bar and containing one line of text, as follows:

Jeffrey Smith 555-7864 47895

A screenshot of a Notepad dialog box with text “MyFriends.txt - Notepad” written in the title bar and containing five lines of text, as follows:

Joe

Rose

Greg

Kirk

Renee

A screenshot of a Notepad dialog box with text “Quotation.txt - Notepad” written in the title bar and containing three lines of text, as follows:

Imagination is more  
important than knowledge.

Albert Einstein

A diagram shows the text written in three lines as follows:

Imagination is more  
important than knowledge.

Albert Einstein

An arrow pointing to the first line is labeled as "Read position."

A diagram shows the text written in three lines as follows:

Imagination is more  
important than knowledge.

Albert Einstein.

An arrow pointing to the second line and is labeled as "Read position."

An arrow points down to a rectangular box containing the text "Open the file with a scanner object." An arrow from the bottom of this box points to a diamond box containing the text "Did hasNext return true?" An arrow labeled "No" extends downward from the bottom of the diamond and points to a box containing the text "Close the file."

An arrow labeled "Yes" extends from the right side of the diamond and points to a rectangular box containing the text "Read an item from file," an arrow points from this box to a rectangular box containing the text "Process the item" and an arrow from this box points to the arrow at the top of the diamond box. An arrow labeled "No" extends from the bottom of the diamond and points to a box containing text "Close the file."

A screenshot of a Notepad dialog box with text “Numbers.txt - Notepad” written in the title bar and containing numbers in five lines, as follows:

8.7

7.9

3.0

9.2

12.6

The diagram shows a box containing number 3 and labeled “instanceCount field (static).”

Below this box is a set of three boxes with the text “object 1,” “object 2,” and “object 3” as instances of the Countable class. From each of these boxes an arrow is pointing to the box at the top.

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: InventoryItem
- Middle row :
  - private description : String
  - private units : int
- Bottom row:
  - public InventoryItem()
  - public InventoryItem(d : String)
  - public InventoryItem(d : String, u : int)
  - public setDescription(d : String) : void
  - public setUnits(u : int) : void
  - public getDescription() : String
  - public getUnits() : int

The diagram shows a box containing text "address" holding the address of the item in a statement "displayItem(item);". An arrow from this box is pointing to "i" of the statement

```
public static void displayItem(InventoryItem i)
{
 System.out.println("Description:" + i.getDescription());
 System.out.println("Units:" + i.getUnits());
}
```

An another arrow from "address" box is pointing to a box labeled "An InventoryItem object" and containing text "description: 'Wrench'; 'units: 20.'"

A diagram shows a box labeled "The item variable holds the address of an InventoryItem object and containing text "address." Below this box, is a box labeled "The i parameter variable holds the address of an InventoryItem object." and containing text: "address."

An arrow from each of these boxes is pointing to a box labeled "An InventoryItem object" and containing text "description: 'Wrench;' 'units: 20.'"

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Dealer
- Middle row :
  - private die1Value : int
  - private die2Value : int
- Bottom row:
  - public Dealer()
  - public rollDice() : void
  - public getChoOrHan() : String
  - public getDie1Value() : int
  - public getDie2Value() : int

The diagram shows rectangular box divided into three rows with the following data filled in:

- Top row: Player
- Middle row :
  - private name : String
  - private guess : String
  - private points : int
- Bottom row:
  - public Player(playerName : String)
  - public makeGuess() : void
  - public addPoints(newPoints : int) : void
  - public getName() : String
  - public getGuess() : String
  - public getPoints() : int

A diagram shows a Method header which reads as follows:

```
public static InventoryItem getData()
```

In this Method header the InventoryItem is the method's return type.

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Stock
- Middle row :
  - private symbol : String
  - private sharePrice : double
- Bottom row:
  - public Stock(sym : String, price : double)
  - public getSymbol() : String
  - public getSharePrice() : double
  - public toString() : String

The diagram shows a box labeled "The company1 variable holds the address of a Stock object." and containing text "address." An arrow from this box is pointing to a box labeled "A Stock object" containing text "symbol: 'XYZ'" and "sharePrice: 9.62."

Below this box is a box labeled "The company2 variable holds the address of a Stock object." and containing text "address" is pointing to a box labeled "A Stock object" containing text "symbol: 'XYZ'" and "sharePrice:9.62."

The boxes containing text "address" is connected by a two way arrow labeled "The if statement compares these two addresses."

A diagram shows a box labeled "The company1 variable holds the address of a Stock object." containing text "address."

Below this is a box labeled "The company2 variable holds the address of a Stock object." containing text "address."

An arrow from each of these boxes is pointing to a box labeled "A Stock object" and containing text "symbol: 'XYZ'" and "sharePrice: 9.62."

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Instructor
- Middle row :
  - private lastName : String
  - private firstName : String
  - private officeNumber : String
- Bottom row:
  - public Instructor(lname : String, fname : String, office : String)
  - public Instructor(object2 : Instructor)
  - public set(lname : String, fname : String,  
◦ office : String) : void
  - public toString() : String"

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: TextBook
- Middle row :
  - private title : String
  - private author : String
  - private publisher : String
- Bottom row:
  - public TextBook(textTitle : String, auth : String,
  - pub : String)
  - public TextBook(object2 : TextBook)
  - public set(textTitle : String, auth : String,
  - pub : String) : void
  - public toString() : String

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: Course
- Middle row :
  - private courseName : String
  - private instructor : Instructor
  - private textBook : TextBook
- Bottom row:
  - public Course(name : String, instr : Instructor,
  - text : TextBook)
  - public getName() : String
  - public getInstructor() : Instructor
  - public getTextBook() : TextBook
  - public toString() : String

A UML diagram titled "Course" contains text as follows:

- Top row: Course
- Middle row :
  - private courseName : String
  - private instructor : Instructor
  - private textBook : TextBook
- Bottom row:
  - public Course(name : String, instr : Instructor, text : TextBook)
  - public getName() : String
  - public getInstructor() : Instructor
  - public getTextBook() : TextBook
  - public toString() : String

Below this UML diagram is a set of two UML diagrams.

The first diagram is the "Instructor" containing text as follows:

- Top row: Instructor
- Middle row:
  - private lastName : String
  - private firstrsltName : String
  - private officeNumber : String

- private courseName : String
- private instructor : Instructor
- private textBook : TextBook
- Bottom row:
  - public Instructor(lname : String, fname : String, office : String)
  - public Instructor(object2 : Instructor)
  - public set(lname : String, fname : String, office : String) : void
  - public toString() : String

The second diagram is the "TextBook" containing text as follows:

- Top row: Instructor
- Middle row:
  - uprivate title : String
  - private author : String
  - private publisher : String
- Bottom row:
  - public TextBook(textTitle : String, author : String, pub : String)
  - public TextBook(object2 : TextBook)
  - public set(textTitle : String, auth : String, pub : String) : void
  - public toString() : String

These three diagrams show the hierarchy with a diamond at the bottom of the "Course" class.

The diagram shows a text box "address" labeled as "The workDay variable holds the address of the Day.WEDNESDAY object." An arrow from this box points to a set of seven boxes arranged in a column containing text from top to bottom as follows:

Day.SUNDAY, Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY, Day.THURSDAY, Day.FRIDAY, Day.SATURDAY. This set of boxes is labeled as "Each of these are objects of the day type, which is a specialized class."

The diagram shows a box labeled "The item1 variable holds the address of an InventoryItem object." and containing text "address."

Below this box is a box labeled "The item2 variable holds the address of an InventoryItem object." and containing text "address."

An arrow from each of these boxes is pointing to a box labeled "An InventoryItem object" containing text "description: 'Wrench'" and "units: 20."

The diagram shows a box labeled “The item1 variable no longer references any object.” and containing text: “null.”

Below this box, is a box labeled “The item2 variable holds the address of an InventoryItem object.” and containing text “address.”

An arrow from this box points to a box labeled “An InventoryItem object” and containing text “description: 'Wrench'” and “units: 20.”

The text below this box reads “This object is still accessible through the item2 variable.”

The diagram shows two rows as follows:

Row1:

A box labeled "The item1 variable no longer references any object." and containing text "null," at the right of this box is a box labeled "An InventoryItem object" and containing text "description: 'Wrench'" and "units: 20." The text below this box reads "This object is inaccessible and will be removed by the garbage collector."

Row2:

A box labeled "The item2 variable no longer references any object." and containing text "null."

The diagram shows a table titled "StockPurchase" with the title labeled as "Name of the class." The data shown in the first column is as follows

Know the stock to Purchase

Know the number of shares to purchase

Calculate the cost of the purchase

This column is labeled as "Responsibilities."

The data shown in the second column is as follows:

Stock class

None

Stock class

This column is labeled as "Collaborating classes."

The first UML diagram is for the “RoomCarpet” with data as follows:

Top row: RoomCarpet

Middle row:

private size : RoomDimension

private carpetCost : double

Bottom row:

public RoomDimension(len : double, w : double)

public getArea() : double

public toString() : String

The second UML diagram is for “RoomDimension” class with data as follows:

Top row: RoomDimension

Middle row:

private length : double

private width : double

Bottom row:

public RoomDimension(len : double, w : double)

public getArea() : double

public toString() : String

There is a diamond at the bottom of the first diagram.

A diagram shows the variable declarations and their memory allocations as follows:

- int count; Enough memory to hold one int. (1234)
- double number; (Enough memory to hold one double. (1234.55))
- char letter; Enough memory to hold one char. (A)

A diagram shows the number variables (single box) pointing towards 'numbers references an array with enough memory for 6 int values labeled as 'Element 0,' 'Element 1,' 'Element 2,' 'Element 3,' 'Element 4,' and 'Element 5.'

A diagram shows ‘Contents of the array after 20 is assigned to numbers[0]’ with ‘numbers[0],’ ‘numbers [1],’ ‘numbers[2],’ ‘numbers [3],’ ‘numbers [4],’ and ‘numbers [5]’ pointing towards an array of six boxes; first box containing ‘20,’ and other 5 boxes containing ‘question mark.’

A diagram shows ‘Contents of the array after 30 is assigned to numbers[3]’ with ‘numbers[0],’ ‘numbers [1],’ ‘numbers[2],’ ‘numbers [3],’ ‘numbers [4],’ and ‘numbers [5]’ pointing towards an array of six boxes; first box containing ‘20,’ fourth box containing ‘30,’ and other boxes containing ‘0.’

A diagram ‘for (int index = 0; index less than NUM\_EMPLOYEES; index plus plus){ System.out.print(“Employee” plus (index plus 1) plus “: ”); hours[index] equals keyboard.nextInt();}’ shows ‘The variable index starts at 0, which is the first valid subscript value’ pointing towards ‘0’ and ‘The loop ends before the variable index reaches 3, which is the first invalid subscript value’ pointing towards ‘NUM\_EMPLOYEES.’

It shows string of 'Subscripts' pointing towards the array as follows:

- 0: 31
- 1: 28
- 2: 31
- 3: 30
- 4: 31
- 5: 30
- 6: 31
- 7: 31
- 8: 30
- 9: 31
- 10: 30
- 11: 31

A diagram shows 'showArray(numbers);' pointing 'array' in 'public static void showArray(int[] array){for (int value : array) System.out.print(value + " ");}' via 'address'. The 'address' points towards an array of 8 elements (5, 10, 15, 20, 25, 30, 35, and 40).

It shows UML diagram with three rows as follows:

- Row 1: Grader
- Row 2: private testScores: double [ ]
- Row 3: public Grader (scoreArray : double[ ]); public plus  
getLowestScore() : double, and public getAverage() : double

It shows 'address' (The names variable holds the address of a String array) pointing towards a 'String array is an array of references to String objects' labeled 'address' as follows:

- names[0] (address) pointing towards “Bill”
- names[1] (address) pointing towards “Susan”
- names[2] (address) pointing towards “Steven”
- names[3] (address) pointing towards “Jean”

It shows ‘address’ (The names variable holds the address of a String array) pointing towards a ‘String array is an array of references to String objects’ labeled ‘null’ as follows:

- names[0] (null)
- names[1] ((null))
- names[2] (null)
- names[3] (null)

It shows ‘address’ (The inventory variable holds the address of an InventoryItem array) pointing towards a ‘The array is an array of references to InventoryItem objects’ labeled ‘null’ as follows:

- inventory[0] (null)
- inventory[1] (null)
- inventory[2] (null)
- inventory[3] (null)
- inventory[4] (null)

It shows 'address' (The inventory variable holds the address of an InventoryItem array) pointing towards an array labeled as 'address' as follows:

- inventory[0] (address) pointing
- inventory[1] (address) pointing
- inventory[2] (address) pointing
- inventory[3] (address) pointing
- inventory[4] (address) pointing

Each of the inventory leads to 'InventoryItem objects' as description: "" and units: 0.

A diagram shows 'Element 0' pointing towards '1,' 'Element 1' towards '7,' 'Element 2' towards '2,' 'Element 3' towards '8,' 'Element 4' towards '9,' and 'Element 5' pointing towards '5' in an array of six boxes. It shows 'These two elements were swapped' pointing towards 'Element 0' and 'Element 5.'

A diagram shows 'Element 0' pointing towards '1,' 'Element 1' towards '2,' 'Element 2' towards '7,' 'Element 3' towards '8,' 'Element 4' towards '9,' and 'Element 5' pointing towards '5' in an array of six boxes. It shows 'These two elements were swapped' pointing towards 'Element 1' and 'Element 2.'

A diagram shows 'Element 0' pointing towards '1,' 'Element 1' towards '2,' 'Element 2' towards '5,' 'Element 3' towards '8,' 'Element 4' towards '9,' and 'Element 5' pointing towards '7' in an array of six boxes. It shows 'These two elements were swapped' pointing towards 'Element 2' and 'Element 5.'

A diagram shows 'Element 0' pointing towards '1,' 'Element 1' towards '2,' 'Element 2' towards '5,' 'Element 3' towards '7,' 'Element 4' towards '9,' and 'Element 5' pointing towards '8' in an array of six boxes. It shows 'These two elements were swapped' pointing towards 'Element 3' and 'Element 5.'

A diagram shows 'Element 0' pointing towards '1,' 'Element 1' towards '2,' 'Element 2' towards '5,' 'Element 3' towards '7,' 'Element 4' towards '8,' and 'Element 9' pointing towards '8' in an array of six boxes. It shows 'These two elements were swapped' pointing towards 'Element 4' and 'Element 5.'

A diagram shows a statement as follows:

“double[ ][ ] scores equals new double[3][4]; ”

In this two sets of empty brackets indicates a two-dimensional array and 3, 4 represents the number of rows and the columns respectively.

**Column 0   Column 1   Column 2   Column 3**

Row 0 scores[0][0] scores[0][1] scores[0][2] scores[0][3]

Row 1 scores[1][0] scores[1][1] scores[1][2] scores[1][3]

Row 2 scores[2][0] scores[2][1] scores[2][2] scores[2][3]

At the top of this table is a box labeled "The scores variable holds the address of a 2D array of doubles." and containing text "address." An arrow from this box is pointing to row 0.

|       | <b>Column 0</b>                                     | <b>Column 1</b>                                     | <b>Column 2</b>                                     | <b>Column 3</b>                                     |
|-------|-----------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------|
| Row 0 | sales[0][0] Holds data for division<br>1, quarter 1 | sales[0][1] Holds data for division<br>1, quarter 2 | sales[0][2] Holds data for division<br>1, quarter 3 | sales[0][3] Holds data for division<br>1, quarter 4 |
| Row 1 | sales[1][0] Holds data for division<br>2, quarter 1 | sales[1][1] Holds data for division<br>2, quarter 2 | sales[1][2] Holds data for division<br>2, quarter 3 | sales[1][3] Holds data for division<br>2, quarter 4 |
| Row 2 | sales[2][0] Holds data for division<br>3, quarter 1 | sales[2][1] Holds data for division<br>3, quarter 2 | sales[2][2] Holds data for division<br>3, quarter 3 | sales[2][3] Holds data for division<br>3, quarter 4 |

At the top of this table is a box labeled: "The sales variable holds the address of a 2D array of doubles." and containing text "address." An arrow from this box is pointing to row 0.

| <b>Column 0</b> | <b>Column 1</b> | <b>Column 2</b> |
|-----------------|-----------------|-----------------|
|-----------------|-----------------|-----------------|

|       |                 |                 |                 |
|-------|-----------------|-----------------|-----------------|
| Row 0 | numbers[0][0] 1 | numbers[0][1] 2 | numbers[0][2] 3 |
|-------|-----------------|-----------------|-----------------|

|       |                 |                 |                 |
|-------|-----------------|-----------------|-----------------|
| Row 1 | numbers[1][0] 4 | numbers[1][1] 5 | numbers[1][2] 6 |
|-------|-----------------|-----------------|-----------------|

|       |                 |                 |                 |
|-------|-----------------|-----------------|-----------------|
| Row 2 | numbers[2][0] 7 | numbers[2][1] 8 | numbers[2][2] 8 |
|-------|-----------------|-----------------|-----------------|

At the top of this table is a box labeled: "The numbers variable holds the address of a 2D array of ints." and containing text "address." An arrow from this box is pointing to row 0.

The diagram shows a box labeled "The numbers variable holds the address of an array of references to arrays." and containing text "address." An arrow from this box is pointing to three boxes arranged in a vertical column labeled "numbers[0]," "numbers[1]," and "numbers[2]" from top to bottom with each containing text "address." An arrow from each of these boxes points to a horizontal array of three boxes.

A diagram shows ‘Lo Shu Magic Square’ with three rows and three columns. The numbers in the first row are 4, 9, and 2. The numbers in the second row are 3, 5, and 7. The numbers in the third row are 8, 1, and 6.

It also shows that the numbers in each row, each column, and each diagonal add up to 15.

The diagram shows two rows as follows:

Row 1: A box containing text "address" and labeled as "The fullName variable holds the address of a string object." An arrow is extending from this box and pointing to another box labeled "A String object" and containing text "Cynthia Susan smith."

Row 2: A box containing text "addresses" and labeled as "The lastName variable holds the address of a string object." An arrow is extending from this box and pointing to another box labeled "A string object" and containing text "Smith."

The diagram shows a box containing text "addresses" and labeled as "The name variable holds the address of a string object," to the right of this box there are two boxes arranged in a column as follows:

A box labeled "This String object is no longer referenced, so it will be removed by the garbage collector." and containing text "George." Below this is a box labeled "The new String objects" and containing text "Sally." An arrow from the "address" box is pointing to the second box in a column.

**A B C D E**

1 87 79 91 82 94 94

2 72 79 81 74 88 88

3 94 92 81 89 96 96

4 77 56 67 81 79 79

5 79 82 85 81 90 90

6

It shows insect (All insects have certain characteristics.) leading to bumblebee (In addition to the common insect characteristics, the bumblebee has its own unique characteristics such as the ability to sting) and grasshoppers (In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump).

It shows UML diagram with three rows as follows:

- Row 1: GradedActivity
- Row 2: private score: double
- Row 3: public setScore(s : double) : void, public getScore() : double and  
public getGrade() : char

It shows UML diagram with three rows as follows:

- Row 1: FinalExam
- Row 2: private numQuestions : int, private pointsEach : double, and private numMissed : int
- Row 3: public FinalExam(questions : int, missed : int), public getPointsEach() : double, and public getNumMissed() : int

The text in the UML diagram for GradedActivity class is as follows:

- Top row: GradedActivity
- Middle row :
  - private score : double
- Bottom row:
  - public setScore(s : double) : void
  - public getScore() : double
  - public getGrade() : char

The text in the UML diagram for FinalExam class is as follows:

- Top row: FinalExam
- Middle row :
  - private numQuestions : int
  - private pointsEach : double
  - private numMissed : int
- Bottom row:
  - public FinalExam(questions : int, missed : int)
  - public getPointsEach() : double
  - public getNumMissed() : int

The UML diagram for the FinalExam class is underneath that of the GradedActivity class with an arrow pointing from UML diagram of

FinalExam class to that of GradedActivity class.

A rectangular box with three rows with the following data filled in:

- Top row: Rectangle
- Middle row:
  - private length: double
  - private width: double
- Bottom row:
  - public Rectangle(len: double, w : double)
  - public setLength(len: double): void
  - public setWidth(w: double): void
  - public getLength(): double
  - public getWidth(): double
  - public getArea(): double

The text in the UML diagram for Rectangle class is as follows:

- Top row: Rectangle
- Middle row :
  - private length : double
  - private width : double
- Bottom row:
  - public Rectangle(len : double, w : double)
  - public setLength(len : double) : void
  - public setWidth(w : double) : void
  - public getLength() : double
  - public getWidth() : double
  - public getArea() : double

The text in the UML diagram for Cube class is as follows:

- Top row: Cube
- Middle row :
  - Private height : double
- Bottom row:
  - public Cube(len : double, w : double, h : double)
  - public getHeight() : double

- public getSurfaceArea() : double
- public getVolume() : double

The UML diagram for the Cube class is underneath that of the Rectangle class with an arrow pointing from UML diagram of Cube class to that of Rectangle class.

A Set of two UML diagrams for the class GradedActivity and CurvedActivity.

The text in the UML diagram for GradedActivity class is as follows:

- Top row: GradedActivity
- Middle row :
  - private score : double
- Bottom row:
  - public setScore(s : double) : void
  - public getScore() : double
  - public getGrade() : char

The text in the UML diagram for CurvedActivity class is as follows:

- Top row: CurvedActivity
- Middle row :
  - private rawScore : double
  - private percentage : double
- Bottom row:
  - public CurvedActivity (percent : double)
  - public setScore(s : double) : void
  - public getRawScore() : double
  - public getPercentage() : double

The UML diagram for the CurvedActivity class is underneath that of the GradedActivity class with an arrow pointing from UML diagram of CurvedActivity class to that of GradedActivity class.

A UML diagram for GradedActivity2 with text in three rows as follows:

- Top row: GradedActivity2
- Middle row:
  - protected score : double
- Bottom row:
  - public setScore(s : double) : void
  - public getScore() : double
  - public getGrade() : char

The diagram shows a UML diagram of "PassFailExam" as follows:

- Top row: PassFailExam
- Middle row:
  - private numQuestions : int
  - private pointsEach : double
  - private numMissed : int
- Bottom row:
  - public PassFailExam(questions : int, missed : int, minPassing : double)
  - public getPointsEach() : double
  - public getNumMissed() : int

This diagram leads to a UML diagram of "PassFailActivity." The UML diagram of "PassFailActivity" is as follows:

- Top row: PassFailActivity
- Middle row:
  - Private minPassingScore : double
- Bottom row:
  - public PassFailActivity(mps : double)
  - public getGrade() : char

This diagram leads to a UML diagram of "GradedActivity." The UML diagram of "GradedActivity" is as follows:

- Top row: GradedActivity
- Middle row:
  - Private score : double
- Bottom row:
  - public setScore(s : double) : void
  - public getScore() : double
  - public getGrade() : char

The "Cube" UML diagram is as follows:

- Top row: Cube
- Middle row:
  - Private height : double
- Bottom row:
  - public Cube(len : double, w : double, h : double)
  - public getHeight() : double
  - public getSurfaceArea() : double
  - public getVolume() : double

Above this UML diagram is a UML diagram of "Rectangle" as follows:

- Top row: Rectangle
- Middle row:
  - private height : double
- Bottom row:
  - public Cube(len : double, w : double, h : double)
  - public getHeight() : double
  - public getSurfaceArea() : double
  - public getVolume() : double

An arrow from "Cube" class is pointing to "Rectangle" class.

The diagram shows a rectangular box divided into three rows with data in each row as follows:

- Top row: Student
- Middle row :
  - private name : String
  - private idNumber: String
  - private yearAdmitted: int
- Bottom row:
  - public Student(n : String, id : String, year : int)
  - public toString() : String
  - public getRemainingHours() : int

A set of three UML diagrams of three classes are as follows:

The UML diagram of "FinalExam3" is as follows:

- Top row: FinalExam3
- Middle row:
  - private numQuestions : int
  - private pointsEach : double
  - private numMissed : int
- Bottom row:
  - public FinalExam(questions : int, missed : int)
  - public getPointsEach() : double
  - public getNumMissed() : int
  - public equals(g : GradedActivity) : Boolean
  - public isGreater(g : GradedActivity) : Boolean
  - public isless(g : GradedActivity) : Boolean

Above this diagram is the UML diagram of "GradedActivity" with data as follows:

- Top row: GradedActivity
- Middle row:
  - Private score : double
- Bottom row:

- public setScore(s : double) : void
- public getScore() : double
- public getGrade() : char

At the right of "FinalExam3" UML diagram, there is another UML diagram for "Relatable interface" with data as follows:

- Top row: <<interface>> Relatable
- Bottom row:
  - public equals (g : GradedActivity) : Boolean
  - public isGreater(g : GradedActivity) : boolean
  - public isLess (g : GradedActivity) : Boolean

A line with a closed arrow head, from "FinalExam3" UML diagram is pointing to "GradedActivity" UML diagram and a dashed arrows from "FinalExam3" UML diagram is pointing to the "Relatable interface" UML diagram.

The statements are as follows:

```
IntCalculator square = new IntCalculator() {
 public int calculate (int number)
 {
 return number * number;
 }};
```

In the code above, “Square” is labeled as “Interface reference variable” and “new IntCalculator()” is labeled as “This creates an instance of an anonymous class that implements IntCalculator.”

The statements below is labeled as ““Method in the anonymous class””:

```
"public int calculate (int number)
{
return number times number;"
```

The hierarchy from top to bottom is shown as follows:

- Object
  - Throwable
    - - Error
    - - Exception
      - IoException
        - - EOFException
        - - FileNotFoundException
      - RuntimeException

The sequence of events is as follows:

```
try
{
 // Create a File object representing the file.
 File equals new File(fileName);

 // Create a Scanner object to read the file.
 // If the file does not exist, the following
 // statement will throw a FileNotFoundException.
 Scanner inputFile equals new Scanner(file);

 // If the file was successfully opened, the
 // following statement will execute.
 System.out.println("The file was found.");
}
catch (FileNotFoundException e)
{
 // If the file was not found, the following
 // statement will execute.
 System.out.println("File not found.");
}

System.out.println("Done.");
```

This diagram depicts that if the statement “Scanner inputFile equals new Scanner(file);” throws an exception, then the lines:

“// If the file was successfully opened, the // following statement will execute. System.out.println("The file was found.");” are skipped.

It also depicts that if the exception is an object of the FileNotFoundException class, the program jumps to “catch (FileNotFoundException e)” clause.

The sequence of events are as follows:

```
try
{
 // Create a File object representing the file.
 File file equals new File(fileName);

 // Create a Scanner object to read the file.
 // If the file does not exist, the following
 // statement will throw a FileNotFoundException.
 Scanner inputFile = new Scanner(file);

 // If the file was successfully opened, the
 // following statement will execute.
 System.out.println("The file was found.");
}
catch (FileNotFoundException e)
{
 // If the file was not found, the following
 // statement will execute.
 System.out.println("File not found.");
}

System.out.println("Done.");
```

An arrow is pointing from the first closed brace to the last statement and is labeled as “If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.”

The hierarchy from bottom to top is as follows:

- NumberFormatException
- IllegalArgumentException
- RuntimeException
- Exception
- Throwable
- Object

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: InventoryItem
- Middle row :
  - private description: String
  - private units: int
- Bottom row:
  - public InventoryItem()
  - public InventoryItem(d : String)
  - public InventoryItem(d : String, u : int)
  - public setDescription(d : String) : void
  - public setUnits(u : int) : void
  - public getDescription() : String
  - public getUnits() : int

The diagram shows a rectangular box divided into three rows with the following data filled in:

- Top row: BankAccount
- Middle row :
  - private balance : double
  - private interestRate : double
  - private interest : double
- Bottom row:
  - public BankAccount(startBalance : double,
  - intRate : double)
  - public deposit(amount : double) : void
  - public withdraw(amount : double) : void
  - public addInterest() : void
  - public getBalance() : double
  - public getInterest() : double

The sequential access shows the data being read from the beginning at the first byte and then to immediately next in the sequence. The random access shows that data can be read in any sequence jumping from one location to the other.

A diagram shows a series of adjacent boxes arranged horizontally. Starting from the left these are labeled as byte number 0, 1, 2 and so on up to byte number 15. A set of two adjacent from the left is labeled as character 0, character 1, character 2 and so on up to character 7.

At the top of the screen are seven selectable tabs: General, Security, Privacy, Content, Connections, Programs, and Advanced. Below are the contents of the currently selected tab, Security. At the top, one of four icons can be selected to view or change security settings: Internet (selected), Local Intranet, Trusted Sites, or Restricted Sites. Below this is the Internet icon (a world globe) and the text "This zone is for Internet websites, except those listed in trusted and restricted zones." At right is a clickable button labeled as Sites. Below this is a moveable slider to set the security level for this zone. It is currently set at Medium-high, with the following annotation:

- Appropriate for most websites
- Prompts before downloading potentially unsafe content
- Unsigned ActiveX controls will not be downloaded

Below is a check box (checked) labeled "Enable Protected Mode (requires restarting Internet Explorer)."

Below are a pair of clickable buttons labeled "Custom level" and "Default level." Below is a clickable button labeled "Reset all zones to default level."

At the extreme bottom there are three clickable buttons: ok, Cancel and Apply.

A screenshot of a command prompt window containing text in three lines as follows:

- First line: Enter the number of hours you worked: 40
- Second line: Enter your hourly pay rate: 50.00
- Third line: Your gross pay is \$2,000.00

The screenshot of the window titled “Wage Calculator” containing two text fields labeled “Number of hours Worked” and “Hourly pay rate.” below these fields is a clickable “Calculate Gross Pay” button. Text fields together are labeled “Text Field controls,” the text filed names are together labeled as “Label Controls,” and the button is labeled as “Button control.”

The hierarchy shown is as follows:

- Root Node
  - Leaf Node
  - Leaf Node
  - Branch node
    - - Leaf Nodes
    - - Leaf Nodes

The screenshots are as follows:

Window 1: It shows text "HBox Demo" in the title bar and three buttons namely: button 1, button 2 and button 3 arranged horizontally from top left corner.

Window 2: It shows text "V8ox Demo" in the title bar and three buttons namely: button 1, button 2 and button 3 arranged vertically from top left corner.

Window 3: It shows text "GridPane Demo" in the title bar and button 1, button 2 and button 5 in the first row from left to right and button 3, button 4 and button 5 in the second row exactly below the buttons in the first row.

A diagram shows the nodes in the scene graph as follows:

- Hbox (Root Node)
  - messageLabel (Leaf Node)

A scene graph for the imageDemo program is as follows:

- hbox (Root Node)
  - imageView (Leaf Node)

A scene graph for the Hboximages program is as follows:

- hbox (Root node)
  - moonIView (Leaf Nodes)
  - shipIView (Leaf Nodes)
  - sunsetIView (Leaf Nodes)

A Scene graph for the VBoximagesWithPadding program is as follows:

- hbox (Root Node)
  - moonIView (Leaf Node)
  - shipIView (Leaf Node)
  - SunsetIView (Leaf Node)

A screenshot of a window with two lines of text in the top-left corner. The text is as follows:

- Line 1:This is label1 This is label2
- Line 1:This is label3 This is label4

A screenshot of a window with a grid of two rows and two columns in the top-left corner. Text in the grid is as follows:

- Row 1 column 1: This is label1
- Row 1 column 2: This is label2
- Row 2 column 1: This is label3
- Row 2 column 2: This is label4

A screenshot of a window shows two rows of two photos:

- Row1: moon and ship
- Row2: sunset and flower

A scene graph for the VBoximagesWithPadding program is as follows:

- gridpane (Root Node)
  - moonIview (Leaf Node)
  - shipIView (Leaf Node)
  - sunsetIView (Leaf Node)
  - flowerIView (Leaf Node)

The layouts shows a box divided into two sections, the first section on the left shows text "Flags of Finland and Germany" enclosed in a box. The box is labeled as "Label." The second section on the right shows flags of Finland and Germany, each labeled as "ImageView" and together enclosed in a box. This box is labeled as "VBox." The outline of the whole layout is labeled as "GridPane with two columns and one row."

A diagram shows a "Click Me" button, labeled "Button Control." An arrow from this button is pointing to the text "ActionEvent Object." An arrow from this text is pointing to the "event" in the first row of following event handler object:

```
void handle(ActionEvent event)
{
 // Event handling code is here...
}
```

The layout shows a box labeled “Vbox” containing another box labeled “Hbox.” Inside this box is a text field labeled “KiloTextField” and text “Enter a distance in kilometers” labeled “promptLabel.” Below HBox is the “Convert” button labeled “CalcButton” and at the bottom is a blank space labeled “resultLabel.”

A scene graph of the kilometer converter application is as follows:

- vbox (Root Node)
  - hbox (Branch Node)
    - - promptLabel (Leaf Node)
    - - kiloTextField (Leaf Node)
  - calcButton (Leaf Node)
  - resultLabel (Leaf Node)

The first screenshot shows the "Kilometer Converter" application with the text field labeled as "Enter a distance in kilometers:" Below is text field is the "Convert" button. The second screenshot shows value entered in the text field as 100 and the result below the "Convert" is shown as 62.14 miles.

The diagram shows a window divided into three horizontal sections. Various regions of the window are shown as follows:

- Top region: Contains menu bar and tool bar.

The Middle region which is further divided into three vertical sections as follows:

- Left Region: Contains navigation panel
- Right region: Contains selection panel
- Center region: Contains various paragraphs and diagrams.
- Bottom region: Contains zoom button and page details.

A screenshot of a window displays three rows with the top row labeled as “This is top.” The middle row is further divided into three parts labeled as “This is left,” “This is center,” and “This is right” from left to right. The bottom row is labeled as “This is bottom.”

A screenshot of a window three rows of cross marks and circles as follows:

- Row 1: Cross mark, Circle, Cross mark
- Row 2: Circle, Cross mark, Cross mark
- Row 3: Circle, Cross, Cross

The text below the third row reads, "X Wins!" and the button "New Game" is shown at the bottom.

The diagram shows a window titled "Slot Machine" containing diagram of three fruits arranged in a row. Below this row is the text field on the left labeled "Amount Inserted: \$" The text on the right reads, "Amount won This spin: \$0.00; Total amount won: \$0.00." The "Spin" button is shown at the bottom.

The screenshot shows a "Metric Converter" window containing text field labeled as "Enter a distance in kilometers:" Below this are radio buttons namely "Convert to Miles," "Convert to feet," and "Convert to inches." Convert to miles is selected. Below these buttons is the "Convert" button.

The screenshots show the "Metric Converter" window with a text field, containing number "100" and labeled as "Enter a distance in kilometers." Below this text field is a set of three radio buttons namely: "Convert to miles," "Convert to feet," and "Convert to inches."

The values displayed by the screenshot on selecting the radio button "Convert to miles," "Convert to feet," and "Convert to inches" are 62.14, 328,100.00, and 3,937,000.00 respectively.

The screenshot shows window with three checkbox buttons, arranged in a column, namely "Pepperoni \$3.00," "Cheese \$2.00," and "Anchovies \$1.00." Below these buttons is the button "Get Total" and below this is the text "\$0.00."

When the first, first two and all three buttons are checked, text below the buttons is displayed as \$3.00, \$5.00, and \$6.00 respectively.

A screenshot of a window shows a list with its components as follows:

- Poodle
- Great Dane
- Labrador
- Terrier

Below this list is another list with a vertical scroll bar as follows:

- Siamese
- Persian
- Bobtail
- Burmese

The first screenshot shows a window containing list as follows:

- Steve
- Bethany
- Will
- Katrina

Below this list is the text "Select a Name" followed by the button "Get the Selection." The second screenshot shows, when "Bethany" is selected it appears below the list in the place of text "Select a Name."

A screenshot of a window shows a list as follows:

- Steve
- Bethany
- Will
- Katrina

Below this list is the text "Nothing Selected" followed by the button "Get the Selection." with the cursor pointing to the button.

A screenshot of a window shows a list as follows:

- Steve
- Bethany
- Will
- Katrina

Below this list is the text "Bethany."

The window shows list as follows:

- January
- February
- March
- April
- May
- June

The first screenshot of a window shows the single selection mode with only item "March" selected. The second shows the multiple selection mode with consecutive items: "February," "March," "April," and "May" selected: The third shows the multiple selection mode with non-consecutive items "January," "February," "March," and "May" selected.

The first screenshot shows the window divided into vertical panels, top panel contains a list, with a vertical scroll bar as follows:

- January
- February
- March
- April
- May
- June

Below this list is an empty panel followed by the button "Get Selections."

The second screenshot shows when February, March, April and June are selected and button is clicked, they appear in the second panel.

The first screenshot shows the window with a panel containing a list and a vertical scroll bar.

The components of list are as follows:

- 1
- 2
- 3
- 4

Below this list is the text "Select Some Numbers" followed by the button "Get the Total."

The second screenshot shows when 2, 3, 4 are selected and the button is clicked, number 9 appears below the list.

The first screenshot shows the window containing the ace of spades, a king, and a queen card, arranged in a row. Below these cards is the text "Select a card." The second screenshot shows that the ace of spades card is selected and the text "Ace of Spades" appears below these cards.

The first screenshot shows the combobox's initial appearance as a rectangular button with an inverted triangle (dropdown button) on extreme right.

The second screen shows a drop down list as follows:

- Wil
- Megan
- Amanda
- Tyler

The mouse cursor is pointing to Amanda.

The third screenshot shows that Amanda is selected.

The first screenshot shows a window with a combobox button with the text “Select a Country” below it followed by the button “Get Selection.”

The second screenshot shows, when drop box button is clicked, a dropdown list appear with the components as follows:

- England
- Scotland
- Ireland
- Wales

The cursor of the mouse is pointing to “Scotland.”

The third screenshot shows that Scotland is selected and it appears below the combobox button replacing text “Select a Country.”

The first screenshot shows a window with a combobox button with text “Select a Country.”

The second screenshot shows, when drop box button is clicked, a dropdown list appear with the components as follows:

- England
- Scotland
- Ireland
- Wales

The cursor of the mouse is pointing to “Scotland.”

The third screenshot shows that Scotland is selected and it appears below the combobox button replacing text “Select a Country.”

The diagram shows a dialog box labeled as “Uneditable ComboBox” containing a combobox with text “The user may only click the ComboBox button and select an item from the dropdown list.” Below this box is a text “Select a Country.”

Below this is a dialog box labeled “Editable ComboBox” containing a blank field with a drop down button, this button is labeled as “Alternatively, the user can click here to select an item from the dropdown list.” and text field is labeled as “The user can type input here.” Below this text field is the text “Select a Country.”

The horizontal slider ranges from 0 to 50, The major tick marks are present at the each increment of 10, starting from 0 and between the major tick marks are five minor tick marks.

The vertical slider has a range of negative 1 to 1. The major tick marks are present at the each increments of 0.25, starting from negative 1 and between the major tick marks are five minor tick marks.

The screenshot shows two text fields labeled “Fahrenheit:” and “Celsius:” containing data “32.0” and “0.0” respectively. Below these fields is a horizontal slider, ranging of 0 through 100, The major tick marks are present at the each increment of 20, starting from 0 and between the major tick marks are five minor tick marks.

The screenshot shows a menu bar containing menus “File” and “Edit.” The edit menu is selected and its menu items list is divided into three sections.

The top section contains Copy, Cut and Paste and they are labeled as “Menu Items.”

The middle section contains Sort and Autosave.

The Autosave is preceded by a check mark box and is labeled as “Check Menu Item.”

The Sort is further divided into Ascending and Descending, which are labeled as “Submenu.”

The bottom section contains two radio buttons namely “Large Font” and “Small Font” and they are labeled as” Radio Menu Items.”

The sections are separated by a horizontal line labeled as “Separator.”

A screenshot of a window shows the “File” and “Text” menus in the menu bar and text “Hello world.” The “Text” menu is selected and its menu items are shown as “Black,” “Red,” “Green,” “Blue,” and “Visible.” The “Black” and “Visible” has check sign before them.

A screenshot shows an open dialog box with drop down button to select the location of file and a search button, the area below these buttons shows a list of files arranged in alphabetical order, at the bottom of the dialog box, there is a text field labeled as “File name:” with open and cancel button below it.

A screenshot shows a save as dialog box with drop down button to select the location of file and a search button, the area below these buttons shows a list of files arranged in alphabetical order, at the bottom of the dialog box, there is text field labeled as “File name:” with save and cancel button below it.

The first set of two screenshots shows the command “java ConsoleDebugging” typed in command prompt window and a “Kilometer Converter” application.

The second set of screenshots shows that user types the value “1000” in the text field of the application and clicked the “Convert” button. The text “621.40 miles” appears below the “Convert” button.

Also, the Debugging messages appear in the command prompt window as follows:

- “Reading 1000 from the text field. Converted value: 1000.000000
- Ready for input.”

The dotted ellipse shows with X-radius equals 150, Y-radius equals 100, and Arc type equals ArcType.ROUND” has a shaded sector with starting angle of 90 degrees and length of its arc is 45 degrees.

Below this ellipse is a dotted circle with X-radius equals 100, Y-radius equals 100, and Arc type = ArcType.ROUND. The circle has shaded sector with starting angle of 0 degrees and length of the arc as 90 degrees.

The first screenshot shows that the circle is initially filled with the default color, black.

The second screenshot shows that when the mouse cursor enters the circle, the circle turns red.

The third screenshot shows that when the user clicks the circle, the circle turns green.

The last screenshot shows that when the mouse cursor exits the circle, the circle turns black.

The diagram shows that the method is first called from the main method of the RecursionDemo class shown as “First call of the method; Value of n: 5. The diagram shows the method being called repeatedly until the value of n becomes 0. The sequence from second through sixth calls is shown as follows:

- Second call of the method; Value of n: 4
- Third call of the method; Value of n: 3
- Fourth call of the method; Value of n: 2
- Fifth call of the method ; Value of n: 1
- Sixth call of the method; Value of n: 0

The diagram shows program code as follows:

```
public static void message(int n)
{
if (n > 0)
{
System.out.println("This is a recursive method.");
message(n - 1);
}
}
```

In the code above, “`message(n minus 1);`” is labeled as “Recursive method call” and the second last brace is labeled as “Control returns here from the recursive call. There are no more statements to execute in this method, so the method returns.”

The diagram shows that the method is first called from the main method and the method is shown as “First call of the method; Value of n: 4; Return value: 24. The diagram shows the method being called repeatedly until the value of n becomes 0. The sequence from second through fifth calls is shown as follows:

- Second call of the method; Value of n: 3; Return value 6
- Third call of the method; Value of n: 2, Return value 2
- Fourth call of the method; Value of n: 1, Return value 1
- Fifth call of the method ; Value of n: 0; Return value 1

It shows original setup as three pegs with first peg containing three discs in the Towers of Hanoi game. It shows the seven moves as follows:

- First move: Move disc 1 to peg 3.
- Second move: Move disc 2 to peg 2.
- Third move: Move disc 1 to peg 2.
- Fourth move: Move disc 3 to peg 3.
- Fifth move: Move disc 1 to peg 1.
- Sixth move: Move disc 2 to peg 3.
- Seventh move: Move disc 1 to peg 3.

A diagram shows the interaction between Java Application and Database Management System shown by a two way arrows between Database Management System and Java Application and manipulation of data by Database Management System is shown by a two way arrow between Database Management System and Data.

A diagram illustrating interaction between Java Application using JDBC classes and Database Management System shown by a two way arrows between Database Management System and Java Application and manipulation of data by Database Management System is shown by a two way arrow between Database Management System and Data. Java Application using JDBC classes is shown by two, two ways arrows between them.

| Description             | ProdNum | Price |
|-------------------------|---------|-------|
| Bolivian Dark           | 14-001  | 8.95  |
| Bolivian Medium         | 14-002  | 8.95  |
| Brazilian Dark          | 15-001  | 7.95  |
| Brazilian Medium        | 15-002  | 7.95  |
| Brazilian Decaf         | 15-003  | 8.55  |
| Central American Dark   | 16-001  | 9.95  |
| Central American Medium | 16-002  | 9.95  |
| Sumatra Dark            | 17-001  | 7.95  |
| Sumatra Decaf           | 17-002  | 8.95  |
| Sumatra Medium          | 17-003  | 7.95  |
| Sumatra Organic Dark    | 17-004  | 11.95 |
| Kona Medium             | 18-001  | 18.45 |
| Kona Dark               | 18-002  | 18.45 |
| French Roast Dark       | 19-001  | 9.65  |
| Galapagos Medium        | 20-001  | 6.85  |
| Guatemalan Dark         | 21-001  | 9.95  |
| Guatemalan Decaf        | 21-002  | 10.45 |
| Guatemalan Medium       | 21-003  | 9.95  |

The third row is labeled as “This row contains data about a single item.  
 Description: Brazilian Dark; Product Number: 15-001; Price: 7.95”

**Description**

Bolivian Dark  
Bolivian Medium  
Brazilian Dark  
Brazilian Medium  
Brazilian Decaf  
Central American Dark  
Central American Medium  
Sumatra Dark  
Sumatra Decaf  
Sumatra Medium  
Sumatra Organic Dark  
Kona Medium  
Kona Dark  
French Roast Dark  
Galapagos Medium  
Guatemalan Dark  
Guatemalan Decaf  
Guatemalan Medium

A ResultSet object contains the results of the SQL query.

Bolivian Dark

Bolivian Medium

Brazilian Dark

Brazilian Medium

Brazilian Decaf

Central American Dark

Central American Medium

Sumatra Dark

Sumatra Decaf

Sumatra Medium

Sumatra Organic Dark

Kona Medium

Kona Dark

French Roast Dark

Galapagos Medium

Guatemalan Dark

Guatemalan Decaf

Guatemalan Medium

A Square box labeled: “Result variable” is pointing to a first row.

### **Column 1**

Row 1 Bolivian Dark  
Row 2 Bolivian Medium  
Row 3 Brazilian Dark  
Row 4 Brazilian Medium  
Row 5 Brazilian Decaf  
Row 6 Central American Dark  
Row 7 Central American Medium  
Row 8 Sumatra Dark  
Row 9 Sumatra Decaf  
Row 10 Sumatra Medium  
Row 11 Sumatra Organic Dark  
Row 12 Kona Medium  
Row 13 Kona Dark  
Row 14 French Roast Dark  
Row 15 Galapagos Medium  
Row 16 Guatemalan Dark  
Row 17 Guatemalan Decaf  
Row 18 Guatemalan Medium

### **Column 1**

|        |                         |
|--------|-------------------------|
| Row 1  | Bolivian Dark           |
| Row 2  | Bolivian Medium         |
| Row 3  | Brazilian Dark          |
| Row 4  | Brazilian Medium        |
| Row 5  | Brazilian Decaf         |
| Row 6  | Central American Dark   |
| Row 7  | Central American Medium |
| Row 8  | Sumatra Dark            |
| Row 9  | Sumatra Decaf           |
| Row 10 | Sumatra Medium          |
| Row 11 | Sumatra Organic Dark    |
| Row 12 | Kona Medium             |
| Row 13 | Kona Dark               |
| Row 14 | French Roast Dark       |
| Row 15 | Galapagos Medium        |
| Row 16 | Guatemalan Dark         |
| Row 17 | Guatemalan Decaf        |
| Row 18 | Guatemalan Medium       |

The text above the Column 1 reads as: “Initially, the cursor is positioned just before the first row in the ResultSet.” and an arrow labeled “Cursor” is pointing.

### **Column 1**

|        |                         |
|--------|-------------------------|
| Row 1  | Bolivian Dark           |
| Row 2  | Bolivian Medium         |
| Row 3  | Brazilian Dark          |
| Row 4  | Brazilian Medium        |
| Row 5  | Brazilian Decaf         |
| Row 6  | Central American Dark   |
| Row 7  | Central American Medium |
| Row 8  | Sumatra Dark            |
| Row 9  | Sumatra Decaf           |
| Row 10 | Sumatra Medium          |
| Row 11 | Sumatra Organic Dark    |
| Row 12 | Kona Medium             |
| Row 13 | Kona Dark               |
| Row 14 | French Roast Dark       |
| Row 15 | Galapagos Medium        |
| Row 16 | Guatemalan Dark         |
| Row 17 | Guatemalan Decaf        |
| Row 18 | Guatemalan Medium       |

The text above the Column 1 reads as: “After the ResultSet object's next method is called the first time, the cursor is positioned at the first row.” and an arrow labeled “Cursor” is pointing to Row 1.

**Description ProdNum Price**

Kona Medium 18-001 18.45

Kona Dark 18-002 18.45

**ProdNum Price**

15-001 7.95

15-002 7.95

17-001 7.95

17-003 7.95

| <b>Description</b> | <b>ProdNum</b> | <b>Price</b> |
|--------------------|----------------|--------------|
| French Roast Dark  | 19-001         | 9.65         |

| <b>Description</b> | <b>ProdNum</b> | <b>Price</b> |
|--------------------|----------------|--------------|
| Brazilian Decaf    | 15-003         | 8.55         |
| Sumatra Decaf      | 17-002         | 8.95         |
| Guatemalan Decaf   | 21-002         | 10.45        |

| <b>Description</b> | <b>ProdNum</b> | <b>Price</b> |
|--------------------|----------------|--------------|
| Galapagos Medium   | 20-001         | 6.85         |
| Guatemalan Dark    | 21-001         | 9.95         |
| Guatemalan Decaf   | 21-002         | 10.45        |
| Guatemalan Medium  | 21-003         | 9.95         |

**Description      ProdNum Price**

Sumatra Organic Dark 17-004 11.95

Guatemalan Decaf 21-002 10.45

| <b>Description</b>      | <b>ProdNum</b> | <b>Price</b> |
|-------------------------|----------------|--------------|
| Bolivian Dark           | 14-001         | 8.95         |
| Brazilian Dark          | 16-001         | 7.95         |
| Central American Dark   | 16-001         | 9.95         |
| Central American Medium | 16-002         | 9.95         |
| Sumatra Dark            | 17-001         | 7.95         |
| Sumatra Organic Dark    | 17-004         | 11.95        |
| Kona Dark               | 18-002         | 18.45        |
| French Roast Dark       | 19-001         | 9.65         |
| Guatemalan Dark         | 21-001         | 9.95         |

It shows two columns; first with ‘Enter a SELECT Query’ heading with blank TextArea and a ‘Submit’ button and second with data in TextArea as:

- 101 Downtown Café 17 N, Main Street Asheville NC 55515
- 102 Main Street Grocery 110 E, Main Street Canton NC 55555
- 103 The Coffee Place 101 Center Plaza Waynesville NC 55516

It shows Entity relationship diagram as follows:

- Customer Table

- CustomerNumber (PK)
- Name
- Address
- City
- State
- Zip
- Phone

- UnpaidOrder Table

- CustomerNumber
- ProdNum
- Quantity
- Cost

- Coffee Table

- Description
- ProdNum (PK)
- Price

It shows CustomerNumber (PK) of Customer Table connected with CustomerNumber of UnpaidOrder Table by a line labeled '1' at left side and

‘infinity’ at right side. It also shows ProdNum of UnpaidOrder Table connected with ProdNum (PK) of Coffee Table by a line labeled ‘infinity’ at left side and ‘1’ at right side.