Programming Fundamentals Using Java

A Game Application Approach

Second Edition

William McAllister
S. Jane Fritz

Programming Fundamentals Using Java

Second Edition

Companion files are also available by writing to the publisher at info@merclearning.com.

Programming Fundamentals Using Java

A Game Application Approach

Second Edition

William McAllister and S. Jane Fritz
St Joseph's College, New York

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai

Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
1-800-232-0223

W. McAllister and S. Jane Fritz. *Programming Fundamentals Using Java: A Game Application Approach*, 2/E.
ISBN: 978-1-68392-669-6

Library of Congress Control Number: 20209528915

212223321   Printed on acid-free paper in the United States of America.

To the memory of my mother Alma, who cherished in me something she was not afforded – a formal education.

—Bill McAllister

To all those who taught me by example that "if you can dream it, you can do it" with gratitude.

—S. Jane Fritz

To our students, whose enthusiasm for learning has always inspired us to pursue improved teaching techniques.

—Bill McAllister
—S. Jane Fritz

Contents

Preface

This is a Java textbook for beginning programmers that uses game programming as a central pedagogical tool to improve student engagement, learning outcomes, and retention. Game programming is incorporated into the text in a way that does not compromise the amount of material traditionally covered in a basic or advanced programming course and permits instructors who are not familiar with game programming and computer graphics concepts to realize the verified pedagogical advantages of game programming.

The book's companion files include a game environment that is easily integrated into projects created with the popular Java Development Environments, including Eclipse, NetBeans, and JCreator in a student-friendly way and also includes a set of executable student games to pique their interest by giving them a glimpse into their future capabilities. The material presented in the book is in compliance with the current ACM/IEEE curriculum guidelines (https://www.acm.org/education/curricula-recommendations), and provides an in-depth discussion of graphical user interfaces (GUIs). It has been used to teach programming to students whose majors are within and outside of the computing fields.

Second Edition Changes

The second edition of the textbook adds features introduced in Java 8 and 9 to the first edition. Specifically, the book now covers streams and the functional programming paradigm, Lambda expressions and their use in streams and in registering event handlers, the expanded features of interfaces and their role in program design, and FX applications.

Chapters 11 and 12, which cover the development of GUI interfaces, are now FX based. The Swing based version of these chapters that were part of the book's first edition, are included on the book's companion disc and posted to the book's companion files FTP site. In addition, pre-Version 8 Java features that were not included in the first edition have been added. These topics include the printf method, conditional expressions, redirecting System.out, and the use of the SecureRandom and StringTokenizer classes.

Features

We use an objects-early approach to learning Java in that the defining and implementation of classes is introduced in the middle of Chapter 3. In preparation for this material, the terms object and class are introduced in Chapter 1 in the context of game piece objects and reinforced in Chapter 2 by continually referring to strings as string objects and differentiating between the primitive types and the String class. In addition, the concept of a reference variable is introduced within the concept of string objects in Chapter 2, and students become familiar with the idea that classes contain data members and methods via the chapter's discussion of the Math class, dialog boxes, and the formatting of numeric values. All of this facilitates the discussion in Chapter 3 of the definition and implementation of methods and classes and the declaration of objects.

The pedagogical tool, game programming, makes the concepts of object-oriented programming more tangible and more interesting to the student. For example, objects are output by drawing them at their current location rather than outputting their (x, y) coordinates to the system console. The functionality of set and get methods and the counting algorithm is illustrated by using them to relocate and animate game piece objects and keep a game's score. Decision statements are used to reflect animated game pieces, detect collisions between them, and to decide when a game is over, and loops are used to draw checkerboard squares and checkers. Because of this new pedagogical approach, student smiles have replaced frowns, enthusiasm has replaced complacency, and "teach us this" has replaced "do we have to know that?" Our classrooms have been transformed from a lecture-based venue to a highly engaged interactive learning environment.

Throughout the book, after a concept is introduced and discussed, its use is illustrated in a succinctly composed working program, and the parts of the program that utilize the new concepts are fully discussed.

Use of the Book

The material in this book can be covered within two courses: a basic programming course followed by an advanced programming course. The basic programming course would normally cover the first seven chapters supplemented with selected materials from Chapters 8 and 10. The remainder of the material would be covered in the advanced course. Alternately, the advanced topics can be incorporated into several other courses such as the use of the GUI chapters in a Web-page-building course, the use of the recursion, generics, and the Application Programming Interface (API) and Collections Framework chapters in a data-structures course, and the multitasking and concurrency chapter in an operating-system course.

The book is written in a way that it and its associated resources could not only be used at the college level, but also at the high school level or used in a self-instructional mode.

Chapter Overviews

Chapter 1: Introduction

This chapter includes a brief history of computer science and topics that are fundamental to an understanding of the concepts presented in the remainder of the textbook. These topics include an overview of the computer system and the representation of data in memory, the programming process and the role of an IDE in that process, platform independence and how Java achieves it, as well as an overview of object-oriented programming and the Application Programming Interface (API). Readers are asked to execute several student-written games contained on the book's companion files, which usually piques their interest, as does the brief description of the game environment included in this chapter.

Chapter 2: Variables, Input/Output, and Calculations

Primitive variables, dialog box input, performing calculations, and performing output to dialog boxes, the system console, and to the game-board window are discussed in this chapter. The declaration of objects and the topic of reference variables are introduced within the context of the declaration of String objects, as are the topics of classes and methods within the chapter's discussion of the Math class, the formatting of text and numeric output, and graphical text output.

Chapter 3: Methods, Classes, and Objects: A First Look

The foundational object-oriented programming concepts used in the next three chapters are discussed in this chapter. It begins with the techniques used to write methods and pass information via value parameters and return statements, and the Graphic class's two-dimensional shape-drawing methods are used in the discussion of parameter passing. The techniques used to specify and write classes are then discussed via a progressively developed game piece class's UML diagram and the progressive implementation of its data members, constructors, and methods. The motivation for set and get methods, and the toString, input, and show methods are discussed and these methods are implemented. Throughout the chapter, sketches are used to illustrate the reference variable and data-member memory model, and the chapter concludes with a graphical application that utilizes the learned concepts.

Chapter 4: Boolean Expressions, Making Decisions, and Disk Input and Output

This chapter begins a two-chapter sequence on control of flow. After a discussion of Boolean expressions and relational and logic operators, the students are introduced to Java's if, if-else, conditional expressions, and switch statements. Their use is illustrated within a graphical context to reflect animated objects, detect when they collide, and to decide which direction to move them in response to a keystroke input. Disk text file I/O is also introduced in this chapter, which is preceded by a discussion of input using the Scanner class and followed by a discussion of redirecting console output, and introduction to the concept and processing of thrown exceptions. The chapter concludes with a graphical application that utilizes the learned concepts.

Chapter 5: Repeating Statements: Loops

The for, while, do-while, and enhanced for loops are presented in this chapter, as are the concepts of counting loops, sentinel loops, and nested loops. The role that the break and continue statements play in repetition constructs is discussed, and Chapter 2's discussion of the formatting of numeric information and the generation of random numbers is extended via a discussion of currency formatting and the API DecimalFormat, Random, and SecureRandom classes. The chapter concludes with a discussion of which loop construct to use for a particular application, and uses a graphical guessing game application and an application that draws a checker board to illustrate these learned concepts.

Chapter 6: Arrays

We placed this chapter after the loops chapter in an effort to immediately reinforce the student's understanding of loops via a discussion of the role loops play in the processing of arrays and the implementation of that processing. The chapter begins with a discussion of the concept of an array and arrays of primitive variables, and it illustrates the primitive array memory model. It then extends these concepts to arrays of reference variables and the objects they reference, and it discusses the passing of arrays to and from methods and illustrates the memory model used to accomplish this. The concept of parallel arrays is discussed as well as the array copying, sorting, minimum, and maximum algorithms and the API implementations of these algorithms. The chapter also discusses multidimensional arrays, and the role arrays play in the addition, and deletion of information contained in disk files. The learned concepts are illustrated within graphical applications that use arrays of game piece objects to display an animated parade and to sort and locate particular game piece objects.

Chapter 7: Methods, Classes, and Objects: A Second Look

This chapter extends the object-oriented programming concepts discussed in Chapter 3 and serves as the OOP foundation on which the remaining chapters of the text are built. It begins with a discussion of static data members, shallow and deep copying and comparisons, and the cloning of objects. The concept of aggregation and its implementation is then discussed, as are inner classes and their methods and the autoboxing feature of the wrapper classes. The processing of large numeric values is also covered in this chapter, as well as enumerated types and the methods of the String and StringTokenizer classes. The learned concepts are illustrated within graphical applications that clone objects, use aggregated game piece objects, parse words from sentences, and perform calculations on large numbers.

Chapter 8: Inheritance

In this chapter, the terminology and concept of inheritance are discussed, as is the way this concept is used in the design and implementation phases of a software project to reduce the time and effort required to complete the project. The topics of extended classes, overriding methods, sub and super classes invoking each other's methods, and the role of abstract and final classes and methods in the design process are also discussed. All of these topics lead into a discussion of polymorphism and polymorphic arrays and the role of polymorphism in the design process. These learned concepts are illustrated in an evolving series of graphical applications that begin with the inheritance of a boat's hull and ends with a polymorphic display of all of the types of boats in a boat dealer's inventory. The chapter concludes with a discussion of interfaces, adapter classes, serialization of objects, and the generation of Javadoc documentation.

Chapter 9 Recursion

This chapter begins by explaining the concept of recursion and recursive methods and a methodology for formulating and implementing recursive algorithms correctly. It then illustrates the use of the methodology in the discovery and implementations of several recursive algorithms, including the Towers of Hanoi. As students progress through the discovery and implementation of these algorithms, they develop the ability to think recursively and to extend the methodology to the discovery and implementation of other recursive algorithms. The chapter concludes with a discussion of the runtime problems associated with recursive algorithms, the role of dynamic programming in the implementation process, and when it is appropriate or efficient to use recursion in the programs we write. The learned concepts are illustrated in applications that compute the terms of the Fibonacci sequence, draw a Sierpinski fractal, and solve the Towers of Hanoi problem.

Chapter 10: Exceptions: A Second Pass

Chapter 4's discussion of catching exceptions thrown from methods we invoke is expanded upon in this chapter, which discusses the *throwing* of exception objects from methods we write. The impact that this has on a method's reusability is discussed and illustrated, as is the ability to create and process exception error messages. In addition, the motivation for creating new exception classes is discussed, as well as the techniques for implementing these classes and using the concept of an exception in a non-error checking mode. The learned concepts are illustrated in several applications that include the use of exceptions in a graphical application to keep a game piece on a game board.

Chapter 11: Graphical User Interfaces

This chapter presents methods used to display enhanced dialog boxes and the fundamental techniques used to incorporate a graphical user interface (GUI) into an FX application. These techniques include the building of an interface that contains two-dimensional shapes and text fields, labels and button components, and the sizing and positioning of these components within the interface with and without the use of a layout manager. The techniques used to write and register event-handler methods that respond to the program user's interaction with these interfaces via mouse actions and keystrokes, and respond to the expiration of timer intervals are also discussed. The learned concepts are illustrated in several applications that use GUIs, and in animations composed without the use of the book's game environment.

Chapter 12: Graphical User Interfaces: A Second Look

The GUI components discussed in Chapter 11 are expanded upon in this chapter to include radio buttons, check boxes, combo boxes, lists, and drop-down and pop-up menus. The chapter also includes a discussion of the use of API dialog boxes that facilitate the specification of a file path to be used in a file I/O operation and the selection of a color to be used in a graphical application. These learned concepts are illustrated in an evolving series of FX based GUI applications that solicit a meal choice from the program user and an application that permits the user to select the background color of the application's window. The chapter concludes with a discussion of Lambda expressions and functional interfaces, and an overview of a GUI drag and drop environment.

Chapter 13: Generics and the API Collections Framework

This chapter begins by introducing the concept of generics and its role in extending the reusability of the methods and classes we write. It discusses the techniques used to implement a generic method that can be passed any type of object and a generic class whose data members' types can be specified when an instance of the class is created. The API Collections Framework is also discussed, which contains a set of generically implemented data structure classes, generic methods that operate on the data stored in these classes, and a set of generic interfaces associated with these classes. These learned concepts are illustrated in a set of applications that implement generic methods and a generic data structure class, and applications that use two of the generic classes in the Collections Framework to store a data set. The chapter concludes with a discussion of streams and the functional programming paradigm.

Chapter 14: Multithreading and Concurrency

The terminology, concepts, advantages, implementation, and problems associated with multithreaded programs are discussed in this chapter. After discussing the implementation of multithreaded applications in Java and the states in which a thread can exist during its lifecycle, our attention turns to the discovery of the problems, including the Producer-Consumer problem, associated with sharing data between threads. Armed with an understanding of these problems, the student is then introduced to the synchronized statement and synchronized methods used to avoid these problems. The chapter concludes with a discussion of the API class, ArrayBlockingQueue, which is used to share data between threads in a problem-free (thread-safe) way. The learned concepts are illustrated in a set of multithreaded applications that share data in an unsafe and safe way and an application that uses an ArrayBlockingQueue instance to share data among threads.

Appendices of the Textbook

The eight appendices contain:

- A description of the game programming environment (Appendix A)
- Directions on how to incorporate the game environment into a programming project (Appendix B)

  Note: The book's companion files contain the game environment and predefined Eclipse, NetBeans, and JCreator project templates that have the game environment incorporated into them.
- An ASCII table that contains the decimal, octal, hexadecimal, and binary representation of each of the characters defined in the table (Appendix C)
- A list of Java keywords (Appendix D)
- A list of all of the Java operators and their precedence (Appendix E)
- A glossary of programming terms (Appendix F)
- A brief description of how to use the API online documentation (Appendix G)
- Answers to the odd numbered Knowledge Exercises that appear at the end of each chapter to facilitate student self-instruction outside the classroom (Appendix H).

The Book's Companion Files

The disc in the back of the book contains a table of contents and the following materials, arranged in separate folders (*also available for downloading by writing to the publisher at info@merclearning.com*):

- Samples of student-written games in an executable format with instruction on how to run them
- The game environment
    - Eclipse, NetBeans, and JCreator template projects with the environment incorporated into them and instructions on how to use them to begin a new project without altering the system's CLASSPATH variable
    - A description of the environment and its call back methods used to draw and animate objects and respond to mouse, keyboard, and timer events
    - The environment's classes and methods in the form of class files, a jar file, and an importable package
- The source files for all of the applications presented in the text
- All of the book's figures
- All of the book's appendices
- The first edition's Swing based versions of Chapters 11 and 12

The Instructor's Files *(available upon adoption to instructors)*

These files contain a table of contents and the following materials, arranged in separate folders:

- Microsoft PowerPoint lecture slides for each chapter
- Solutions to all of the programming exercises that appear at the end of each chapter
- Answers to all of the knowledge exercises that appear at the end of each chapter
- A capstone game project assignment for a basic Java course
- Samples of student-written games in an executable format with instruction on how to run them
- Sample quizzes and final examinations
- Sample Syllabi for a basic and advanced Java course
- Samples of high school programming competition problems
- The first edition's Swing based versions of Chapters 11 and 12

Digital Versions

Digital versions of this text and its resources are available on the publisher's electronic delivery site, academiccourseware.com, as well as other popular e-vendor sites.

W. McAllister
S. Jane Fritz
Patchogue, NY
March 2021

Acknowledgments

We would like to thank the instructors and students who have used the 1st edition of this textbook, for their feedback that has been incorporated into, and inspired, this 2nd edition. We would also like to thank our students who designed and implemented the game programs on the companion files that accompany this edition, as a semester long project in their basic Java course.

We would also like to thank the administration of St. Joseph's College and the members of the Promotion and Awards Committee for granting Bill a sabbatical, which was dedicated to the preparation of the 1st edition's manuscript, as well the entire St. Joseph's College community who offered encouragement and support during the preparation of both the 1st and 2nd editions.

We also thank David Pallai, the publisher and founder of Mercury Learning and Information for establishing and managing a publishing company that produces a high-quality product at an affordable price, and his production team for guiding us through the development and production process, specifically Jennifer Blaney, Meg Salvia, and Patrick Smith.

Finally, we would like to thank our family and friends for their endless patience when we were too busy to be ourselves.

To the Students

It is our hope that our approach to the material in this book will challenge you, engage you, and inspire you to continue your study of computer science and to enjoy a rewarding career by immersing yourselves in this area of national need.

Credits

Chapter 1

Computer © Robert Lucian Crusitu/Shutterstock.com, Image ID: 156076811

Figure 1.5 Abacus, by Gisling (Own work) [CC-BY-3.0 (http://creativecommons.org/licenses/by/3.0)], via Wikimedia Commons), (http://upload.wikimedia.org/wikipedia/commons/d/d4/Positional_decimal_system_on_abacus.JPG)

Figure 1.6 Slide rule, by Ricce (Own work) [Public domain], via Wikimedia Common,

(http://upload.wikimedia.org/wikipedia/commons/9/98/Regolo_calcolatore.jpg)

Figure 1.7 Blaise Pascal, by Mahlum (Own work) [Public domain], via Wikimedia Commons, (http://upload.wikimedia.org/wikipedia/commons/4/4d/Pascal_Blaise.jpeg)

Figure 1.8 Jacquard's Loom,

(http://upload.wikimedia.org/wikipedia/commons/6/65/Jacquard_loom.jpg)

Figure 1.9 Charles Babbage, (http://upload.wikimedia.org/wikipedia/commons/1/1d/Charles_Babbage_Difference_Engine_No1.jpg)

Figure 1.10 Ada, Margaret Sarah Carpenter [Public domain], via Wikimedia Commons (http://commons.wikimedia.org/wiki/Ada_Lovelace#mediaviewer/File:Carpenter_portrait_of_Ada_Lovelace_-_detail.png)

Figure 1.11 Hollerith (https://www.census.gov/history/img/HollerithMachine.jpg)

Figure 1.12 Turing Statue © Guy Erwood/Shutterstock.com

Figure 1.13a. Troubleshooting the ENIAC (http://ftp.arl.army.mil/ftp/historic- computers/jpeg/eniac3.jpg)

Figure 1.13b. Troubleshooting the ENIAC (http://ftp.arl.army.mil/ftp/historic- computers/jpeg/eniac1.jpg)

Figure 1.14a. Programming the ENIAC (http://ftp.arl.army.mil/ftp/historic-computers/jpeg/first_four.jpg)

Figure 1.14 b. Programming the ENIAC (http://ftp.arl.army.mil/ftp/historic-computers/gif/eniac4.gif)

Figure 1.15 John von Neumann, (Public domain), (http://commons.wikimedia.org/wiki/File:JohnvonNeumann-LosAlamos.gif)

Figure 1.16 Grace Hopper (http://www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm)

Figure 1.17 Grace Hopper's Bug (http://www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm)

Figure 1.18 Steve Jobs, by Kees de Vos from The Hague, The Netherlands [CC-BY-SA-2.0 (http://creativecommons.org/licenses/by-sa/2.0)], via Wikimedia Commons, (http://upload.wikimedia.org/wikipedia/commons/5/54/Steve_Jobs.jpg)

Figure 1.19 Bill Gates, by Matthew YoheAido2002 at en.wikipedia [CC-BY-3.0 (http://creativecommons.org/licenses/by/3.0)], from Wikimedia Com, (http://upload.wikimedia.org/wikipedia/commons/7/7f/Bill_Gates_2004_cr.jpg)

Figure 1.20 Vinton Cerf and Robert Kahn (http://georgewbush-whitehouse.archives.gov/ask/20051109-2.html)

Figure 1.21 Donald Knuth (Case Alumni Association and Foundation 2010, Flicker and (http://www.casealum.org/view.image?Id=1818)

Figure 1.22 Tim Berners-Lee, Courtesy of World Wide Web Consortium, Massachusetts Institute of Technology (www.w3.org/People/Berners-Lee)

Chapter 3

Green tea cup on windowsill © GoodMood Photo/Shutterstock.com, Image ID: 158310728

Chapter 4

Two Game Figurines © Melanie Kintz, Mellimage/Shutterstock.com, Image ID: 70027117

Chapter 5

Loops of a scaring roller coaster © Marcio Jose Bastos Silva/Shutterstock.com, Image ID: 97819241

Chapter 8

Boats on phewa-lake-nepal © Worapan Kong/ShutterStock.com, Image ID: 132349601

Chapter 9

Nested Traditional Matryoschka Dolls © PiXXart/Shutterstock.com, Image ID: 112054022

Chapter 10

Construction site © Victor Correia/ShutterStock.com, Image ID: 113180041

Chapter 11

Pocket sliding fifteen puzzle game © Coprid/ShutterStock.com, Image ID: 81197962

Chapter 12

Vector Retro Menu Design © Yienkeat/ShutterStock.com, Image ID: 108289445

Chapter 13

Collection of spaceship, planets and stars © Motuwe/Shutterstock.com, Image ID: 140336917

Chapter 14

Maze Game with Solution © VOOK/ShutterStock.com, Image ID: 95912809

Chapter 1

Introduction

In this chapter

This chapter presents topics that are fundamental to computing and the programming process and discusses tools that programmers use to write programs. Because the focus of this text is on learning to program, an understanding of these concepts and tools is essential. The topics include a brief history of computing, which will highlight some of the important contributions to the field and facilitate an understanding of the modern computer system as well as how data is stored. The tools discussed in the chapter are used to develop an unambiguous description of a program and to minimize the effort required to transform this description into a functional program that can run on any computer system or mobile device.

After successfully completing this chapter, you should:

- Understand the hardware and software components of a computer system
- Gain an appreciation for the history and evolution of computing
- Be able to specify simple programs and games
- Have used some examples of student-written game programs
- Understand why Java programs can be run on any computer system
- Be familiar with the concept of objects, classes, and the object-oriented programming paradigm
- Understand the programming process and the role of Integrated Development Environment programs in this process
- Be familiar with the features of the game-development tool on the CD that accompanies this textbook
- Understand how data is represented inside computer systems

1.1 THE COMPUTER SYSTEM

Over the last twenty years, computers and the use of the Internet have become part of our everyday lives. Daily communication that was performed using postal systems and telephone conversations are now performed more efficiently using computer-based e-mails and text messaging. Much of the information gathering we performed in libraries is now done from the comfort of our homes using a computer attached to the Internet, as is much of the shopping we do. As a result, the number of computers available in the world continues to grow (Figure 1.1).

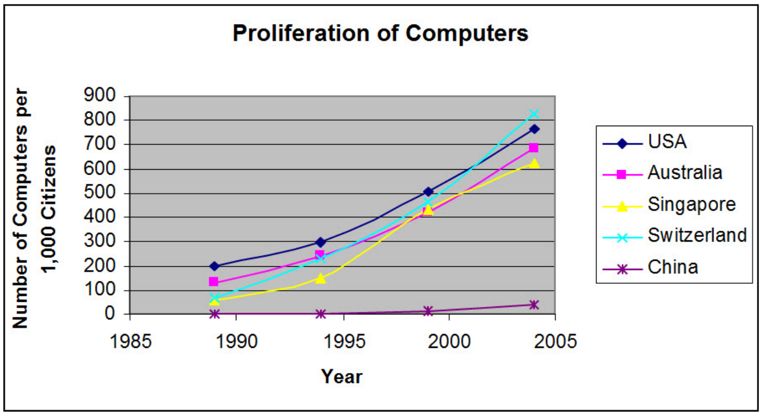


Figure 1.1
Growth in the number of computers per capita over a fifteen-year period.

In many of the developed countries of the world there is now one computer, or more accurately, one computer system for every citizen in the country. Although many of these people would say that they use a computer every day, they really should say that they use a computer system every day.

As shown in Figure 1.2, a computer system is comprised of two major components: software and hardware. As its name implies, hardware is the hard, or tangible, part of the computer system. It is the collection of electronic circuits, mechanical devices, and enclosures manufactured in a factory. When we purchase a computer system and look into the box it comes in, what we see is the hardware.

However, the box also contains software, but, as its name implies, it is the soft, or less-tangible portion of the computer system, and so it is not as easy to detect. Software, or programs, consists

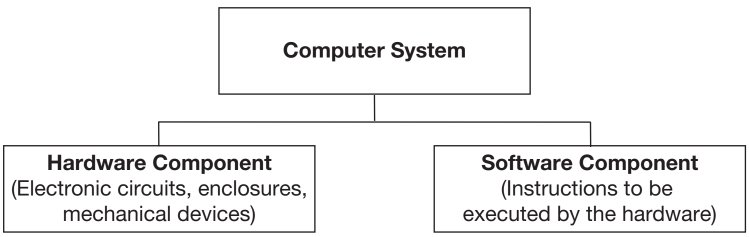


Figure 1.2
The two major components of a computer system.

of sequences of instructions written by programmers to perform specific tasks. These instructions are executed by the computer system's hardware, and both components are essential to a computer system. A computer system that contained only hardware would have no instructions to execute, so it would do nothing but consume electrical power. A computer system that contained only software would not be able to execute the program's instructions.

The software of a computer system is comprised of two major subcomponents: operating system software and application software (Figure 1.3). Microsoft Windows, Apple OS, and Linux are all examples of operating system programs. This set of programs contains instructions to manage the hardware resources of the computer system and provides an interface, usually a point-and-click interface, through which the user interacts with the computer system. In addition, most application software interacts with the hardware through various groups of operating system instructions.

Although nonoperating system software can be categorized in several groupings, we will consider all nonoperating system software to be collected into one group, application software, as shown in Figure 1.3. In this textbook, we will learn how to write application software using the programming language Java.

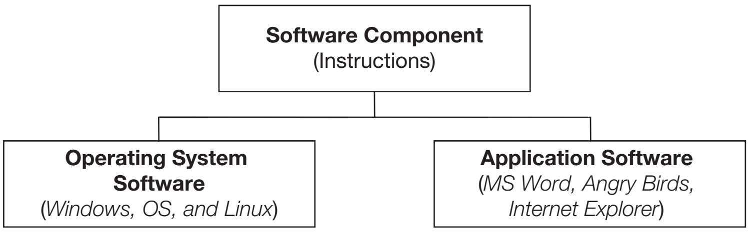


Figure 1.3
Computer software subcomponents.

The hardware of the computer system can be divided into three main categories, or subcomponents, based on the function they perform. Hardware that communicates with humans and other computer systems is grouped into the category input/output (I/O) devices. Hardware used to store information inside the computer system is grouped into the category storage devices. Finally, all other functions performed by the hardware are part of a category named the central processing unit (CPU).

Figure 1.4 shows the standard conceptual arrangement of the three hardware categories with the CPU at the center of the arrangement. The storage devices are shown on the right and bottom of the figure and are divided into two types of devices: backing storage, often referred to as secondary storage, and random access memory (RAM), also called main memory. With the exception of backing storage, each of the hardware components has been assigned an acronym, which is shown parenthetically below the name of the component in Figure 1.4. For brevity, the components are most often referred to using their acronyms: I/O devices, CPU, and RAM (pronounced as "ram").

The arrows into and out of the I/O devices at the top left of Figure 1.4 indicate the flow of information entering (input) and leaving (output) the computer system. The other arrows in the

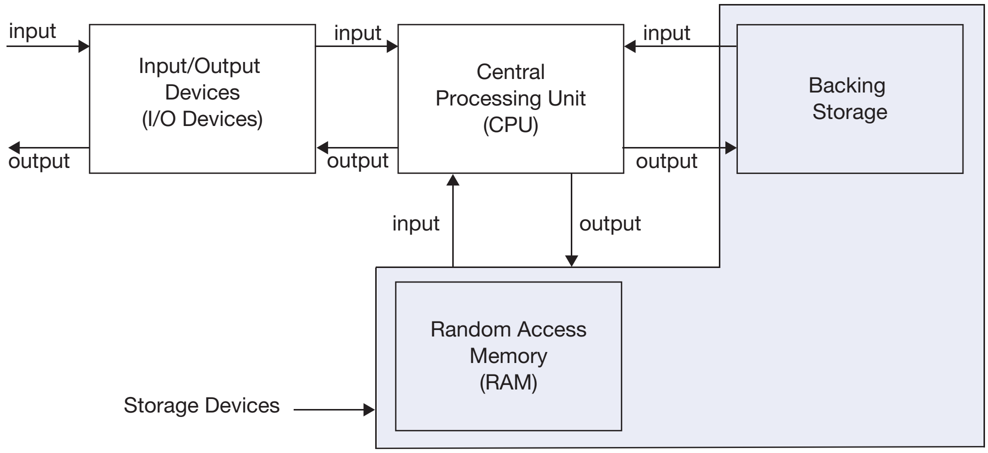


Figure 1.4
Arrangement of the hardware subcomponents of a computer system.

figure represent the flow of information among the computer-system components. The central processing unit can receive information from (arrows labeled "input") and send information to (arrows labeled "output") the other system components. The flow of information is always relative to the CPU. Information sent to the CPU is considered input, and information sent from the CPU is considered output.

Hardware that communicates with humans and other computer systems is grouped into the component I/O devices. These devices are the interface between the computer system and the rest of the world. Input devices send information into the system. Examples of input devices include a keyboard, a touch screen, a mouse, a microphone, a digitizer, and a modem. Output devices send information out from the system. Examples of output devices include a monitor, a printer, a speaker, and a modem. A modem is both an input and output device and is normally used to transfer information between computer systems.

Hardware devices that have the ability to store and recall information are grouped into the component storage services. All but one of these devices fall into the subcomponent backing storage, shown on the top right side of Figure 1.4. Examples of storage devices include hard drives, flash drives, subscriber identification module (SIM) cards, CD drives, and magnetic tape drives. One storage device, random access memory (RAM) is depicted separately at the bottom of Figure 1.4. One difference between this storage device and all of the other storage devices is its speed. It can access information, meaning store and recall information, faster than any other storage device. Its information-access speed approaches the speed at which the CPU can transfer information.

Programs run faster when their instructions are stored in RAM, so it is an advantage to have a high-capacity RAM in a computer system. Unfortunately, the materials and manufacturing process used to achieve RAM's speed make it the most expensive type of storage. To make computer systems affordable, backing-storage devices are added to the system. When a program is in execution, the operating system software attempts to transfer the program's instructions and the data the instruction processes from backing storage to RAM before this information is needed by the CPU.

Another reason for adding backing storage, or secondary storage, to a computer system is the fact that RAM is volatile, which means that it only retains its memory when it is attached to an electrical power source: no electricity, no memory. All backing-storage devices are nonvolatile, which means that the information they store is not lost when they are detached from a power source. As a result, these devices can be used to archive program instructions and data within the computer system when it is powered down (e.g., hard drives), and can be used to manually transport information between computer systems (e.g., flash drives).

The I/O and storage components of the computer system give us the ability to transfer information into and out of the computer system and the ability to store and recall that information. The CPU depicted in the center of Figure 1.4 gives us the ability to process the information, and so it is aptly named the central processing unit. If we were inclined to designate one of the computer-system components as the brain of the system, we would probably bestow that title on the CPU. However, despite the remarkable tasks that computers perform, the CPU's electronic circuits only perform five very basic processing operations:

1. Transfer information (i.e., instructions and data) to and from the other components of the computer system and interpret instructions
2. Store a very small amount of information, e.g., one instruction and sixteen pieces of data
3. Perform arithmetic operations such as addition, subtraction, multiplication, and division
4. Perform logic operations involving relational operators (such as $10 < 6$, and $a >= 12$) and logical operators (such as A AND B, and A OR B)
5. Execute instructions in the order in which they are written or skip some instructions based on the truth value of a logic operation

The magic here is that all of the remarkable tasks that computers do have been expressed as a sequence of these five basic processing operations. A step-by-step sequence of these operations to perform a particular task is called an **algorithm**. The most difficult part of a programmer's job is to develop, or discover, algorithms. Once an algorithm is discovered, it is written into a programming language and verified via a testing process.

Definition

An **algorithm** is a step-by-step sequence of the five processing operations a computer system can execute to solve a problem or perform a particular task.

A **computer program** is an algorithm written in a programming language.

1.2 A BRIEF HISTORY OF COMPUTING

Long before our modern computers existed, people had the need to count or compute. As a matter of fact, the early meaning of the term computer referred not to a machine but to a person who performed calculations. In this section, we will see the amazing development of the revolutionary machines that have changed the way we learn, teach, shop, do research, and are entertained.

1.2.1 Early Computing Devices[1]

If computers are really such an important part of our lives today, you might wonder and ask the question: Who invented the computer?

Although this is a simple question, it does not have a simple answer such as Thomas Edison invented the light bulb or Alexander Graham Bell invented the telephone. One reason for this complexity is that computers evolved over thousands of years, and many people from different cultures and diverse fields such as mathematics, physics, engineering, business, and even textile design were involved in laying the foundation for the modern electronic computer.

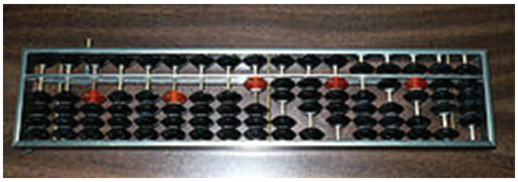


Figure 1.5
The abacus.

Figure 1.6
A typical modern slide rule.

The roots of computing dates from about 50,000 to 30,000 BC when people counted their sheep and other possessions using their fingers, stones, or notches on sticks. The first computing device, the abacus, was introduced in China around 2,600 BC and used pebbles or stones. A later version of the abacus (shown in Figure 1.5) used beads that could be moved on a wire frame to perform basic counting and arithmetic functions. These were widely used in Europe and Asia, and some of these devices are still in use today.[2]

It was not until the seventeenth century that there were other notable attempts at building computing devices. Napier's bones and the slide rule (Figure 1.6) were two of these devices.

Figure 1.7
Blaise Pascal: philosopher, mathematician, inventor.

Blaise Pascal (Figure 1.7), at the age of 18, built a mechanical calculator called the Pascaline to perform basic addition and multiplication.

Because manufacturing technology was not yet well developed, these devices had to be carved or forged by hand, which required tedious work. Although it would seem likely that the development of computing devices would continue at a more rapid pace, very little progress was made from the seventeenth century until the 1800s, and we might ask why. Perhaps it was because this was a time of war, colonization, and the struggle for survival in much of the world. (If you think about the United States, for example, from 1776 through the 1800s, building a calculator was not considered a priority at that time.)

Figure 1.8
A Jacquard Loom.

In 1801, Joseph Marie Jacquard, a textile designer, discovered that he could program his weaving loom (Figure 1.8) to create intricate patterns in the fabric, by storing the instructions on punched cards or paper tape. These binary instructions directed the loom to raise or lower certain threads depending upon whether or not a hole was punched on the tape. Later on, this concept would develop into the idea of creating a stored program computer based on binary instructions; it would be implemented in the twentieth century using punched cards for computer input.

1.2.2 Computers Become a Reality

Charles Babbage, a mathematician working in England around 1822, designed the prototype of a machine, known as the Difference Engine, to compile mathematical tables. It was a large hand-cranked machine built of metal wheels and gears and although he continued to add refinements to it, he never fully completed it. By 1837, Babbage took his ideas one step further and designed a more complex Analytical Engine Figure 1.9), which he envisioned to be a general purpose computational machine and which had many characteristics in common with modern computers. He designed it to be steam powered, which would not make it portable, but would automate mathematical calculations. Due to the limitations of available technology, it was not completed in his lifetime, but it has recently been completed and works as he described it. Charles Babbage has been called the father of the computer for his innovative work on the first mechanical computer.

Lady Ada Augusta Byron Lovelace (Figure 1.10), the daughter of the poet Lord Byron, became intrigued with Babbage's work and began to write instructions, or what we now call programs, for his machine. She is known today as the first programmer, and the programming language used for U.S. government applications is named Ada in her honor. Ada was unique in being a well-educated woman, skilled in mathematics, at a time when women had little formal or advanced schooling. She was able to perform the advanced mathematical and engineering design functions required for programming a theoretical computer that was not yet completely operational.

In the 1890s, in the United States, Herman Hollerith was working on a mechanical calculator and was asked by the government to design a machine that could record and store census data. The population was growing so fast that hand calculation could not keep pace with the growing volume of data: by the time the data was tabulated it was outdated and the next census had begun. Hollerith used punched cards to input the data to his new machine (Figure 1.11), which successfully compiled and tabulated even greater amounts of data in record time. Following this success, he founded a company with Thomas Watson, which later became known for computing, the International Business Machine Corporation (IBM).

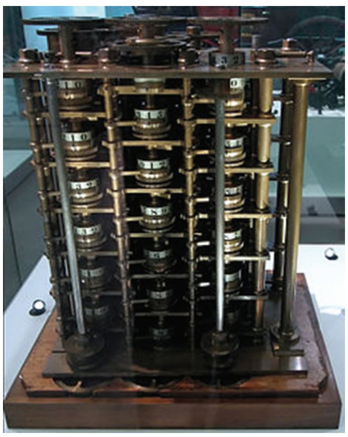


Figure 1.9
Charles Babbage's Analytical Engine.

Figure 1.10
Lady Ada Augusta Byron Lovelace, the first programmer.

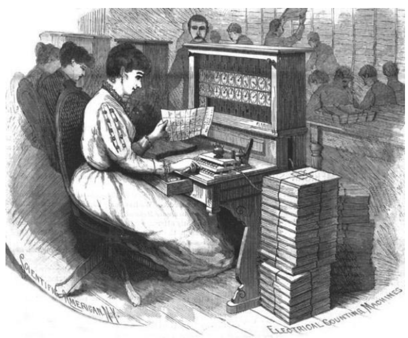


Figure 1.11
Hollerith's electric tabulating machine.

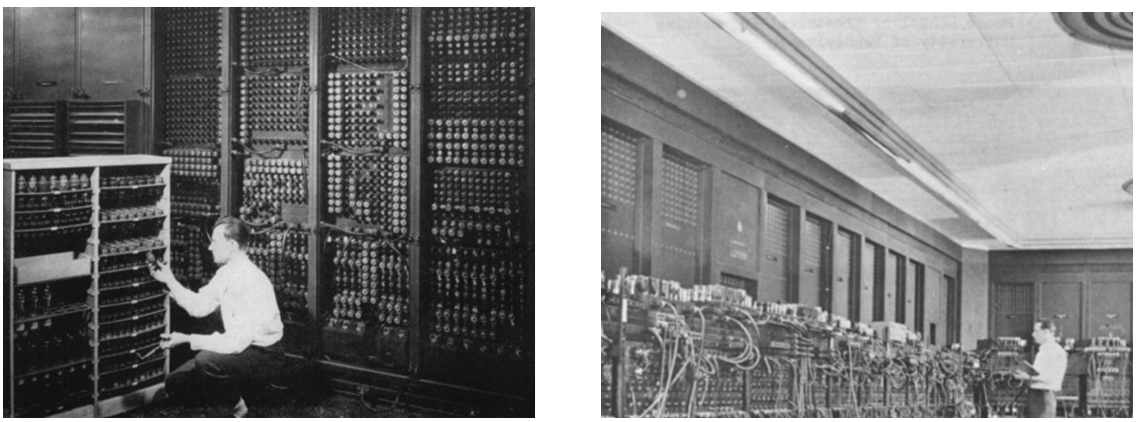
Figure 1.12
Alan Turing, father of theoretical computer science.

In the 1900s, the demand for recording and processing large amounts of data continued to increase, and there were numerous attempts to design more advanced computing machines. The need came from businesses as well as the military. Large universities and mathematicians throughout the world began to design and build these early computing machines. Around 1939–1942, the Atanasoff-Berry computer (ABC) was built by Dr. John V. Atanasoff and Clifford Berry at Iowa State University. It was the first electronic digital computer. At about the same time, Konrad Zuse, working in Germany, built the first fully programmable computer, the Z3.

Also in the early 1940s, the Colossus was built with the assistance of the brilliant British mathematician Alan Turing (Figure 1.12). It was designed as a code-breaking machine that could decipher the German codes created with the Enigma encoding machine. Turing's contribution to breaking the German codes helped to defeat Hitler in World War II. Turing also explored Artificial Intelligence (Google "Turing Test" for specific details), and he is highly regarded as the father of theoretical computer science, laying a foundation upon which to build advanced computing machines.

In 1944, the Harvard Mark I was designed and built through the efforts of Howard Aiken working with Grace Hopper. Built at Harvard University by IBM, the Mark I was the first electromechanical computer, and it was used to produce mathematical tables. It could be programmed using paper tape.

The first electronic general purpose digital computer, the Electronic Numerical Integrator and Computer (ENIAC) was built at the University of Pennsylvania in 1946 by John Mauchly and J. Presper Eckert. This computer weighed 30 tons and had over 18,000 vacuum tubes and thousands of electronic relays (Figure 1.13). It filled a large room that was required to be air conditioned because of the heat this machine generated. It could add or subtract 5,000 times a second, a thousand times faster than any other machine at that time. It also had modules to multiply, divide, and calculate square roots.



(a) Replacing vacuum tubes.

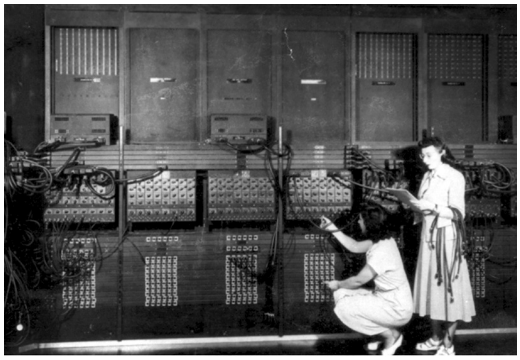
(b) Programming the ENIAC.

U.S. Army photos.

Figure 1.13
Troubleshooting the ENIAC.

Most of the ENIAC's programming was done by six women, including those shown in Figure 1.14.

U.S. Army photo



U.S Army photo

Figure 1.14
First programmers of the ENIAC.

Figure 1.15
John von Neumann, father of the modern computer and game theory.

John von Neumann (Figure 1.15) proposed modifications to the ENIAC, which included using binary instead of decimal numbers. His design for a stored program binary computer where both the program and the data could be stored in the computer's memory became known as the von Neumann architecture, which is still in use today. In 1945, he proposed the design for the Electronic Discrete Variable Automatic Computer (EDVAC) and later worked on the Institute for Advanced Study (IAS) computer in Princeton. He is often called the father of the modern computer and game theory.

1.2.3 Computer Generations[3]

The computers that followed are usually grouped into generations, each characterized by a specific component or technology. The dates are approximate.

First-Generation (1937–1946): *Vacuum Tubes*

These very large computers used thousands of vacuum tubes, generated a lot of heat, and were fairly unreliable. Memory storage was on magnetic drums, input was performed using punched cards or paper tape, and output was displayed on paper printouts. Computers of this generation could only perform a single task, lacked an operating system, and were programmed using a sequence of ones and zeros known as machine language. First-generation machines include the ENIAC, Electronic Delay Storage Automatic Calculator (EDSAC) and EDVAC computers.

Second-Generation (1947–1963): *Transistors*

This generation of computers used transistors, which were much more reliable than the vacuum tubes they replaced. Transistors were also smaller, cheaper, and consumed less electrical power. Machine language was replaced with assembly language, which was a more English-like language, and higher-level languages such as Common Business Oriented Language (COBOL) and Formula Translation (FORTRAN) were developed for this generation of computers. In 1951, the universal automatic computer (UNIVAC 1) was introduced as the first commercial computer. In 1953, the IBM 650 and 700 series computers were introduced. Operating systems were designed for these machines, and over 100 computer-programming languages were developed during this generation. Storage media such as magnetic tape and disks were in use, and printers were available for output.

Third-Generation (1964–1971): *Integrated Circuits (IC) or "chips"*

Transistors were miniaturized and placed on chips and integrated circuits (IC), developed by Jack Kilby and Robert Noyce. This invention resulted in smaller, more powerful, more reliable, and cheaper computers. Users could now interact with computers through keyboards and monitors instead of punched cards and printouts. Operating systems monitored memory usage and controlled the scheduling of multiple applications that could share the system resources.

Fourth-Generation (1971-present): *Microprocessors and Very Large Scale Integration (VLSI)*

Very-large-scale integration (VLSI) resulted in thousands of computer circuits being reduced to fit on a chip, reducing the room-size computers of the first generation to something that could fit in your hand. Components of the computer, from the central processing unit and memory to input/output controls, could now be located on a single microprocessor chip. In addition to their small size, computers became affordable for individuals, and in 1977, the personal computer (PC) became available from three companies: Apple, Tandy/Radio Shack, and Commodore. In 1980, Microsoft released its disk operating system (MS-DOS), and in 1981, IBM introduced the PC for home and office use. Three years later, Apple introduced the Macintosh computer with its icon-driven interface. In 1985, Microsoft released the Windows operating system. Fourth-generation computers also used graphical user interfaces (GUI, pronounced "gooey") and provided a mouse for ease of use. Object-oriented languages, such as Java, were developed for more efficient software development. These smaller, more reliable and powerful computers could now be linked together, resulting in the growth of networks and the Internet.

Fifth-Generation (Present and Beyond): *Artificial Intelligence, Parallel Processing, Quantum Computing*

Fifth-generation computing devices are characterized by artificial intelligence and the advancement of devices that will respond to natural language and be capable of learning. Although these features are still in the early stages of development, some applications such as voice recognition are currently available. The use of parallel processing, quantum computing, and nanotechnology will help to achieve these advances and will change computing in the future.

1.2.4 More Notable Contributions

Figure 1.16
Admiral Grace Hopper.

In addition to the achievements already mentioned, there were many others who made notable contributions to the computing field. The names and contributions of a few of these innovators follow, and you are invited to continue to add to the list.

Admiral Grace Murray Hopper (Figure 1.16) was a pioneer in the field of computing. She was one of the first programmers of the Harvard Mark I computer and is known for the development of the first compiler and assembly language. Her work in programming led to the development of the language COBOL, and she later worked on Ada.

She coined the term "debugging" when she removed a bug (or moth) from a computer's circuitry that was interrupting the flow of electricity, and taped it into her notebook (Figure 1.17).

Steve Jobs (Figure 1.18) and Steve Wozniak were the cofounders of Apple Computers. The Apple I was one of the three personal computers introduced in 1977 for home use. Together they developed the point-and-click approach to computing. In 1984, they introduced the MAC OS that developed into the modern graphical user interface, which today is standard on modern computers. Steve Jobs is also a cofounder of Pixar Animation and has been described as the father of the digital revolution.

Bill Gates (Figure 1.19) and Paul Allen cofounded Microsoft, one of the largest U.S. corporations, and supplied the disk operating system (DOS) to IBM to run on its PCs. In 1985, Microsoft developed a graphical operating system known as Windows, which is the operating system used on over 80% of today's computers.

James Gosling is credited with the development of the object-oriented programming language known as Java. He is called the father of Java programming.

Bob Metcalfe and David Boggs invented the Ethernet, the technology upon which local computer area networks are based.

Vinton Cerf (Figure 1.20a) and Robert Kahn (Figure 1.20b) are considered to be the fathers of the Internet and the Transmission Control Protocol/Internet Protocol (TCP/IP) upon which the

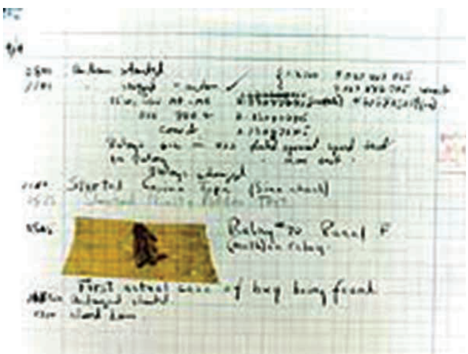


Figure 1.17
Grace Hopper's first recorded computer "bug."

Figure 1.18
Steve Jobs, cofounder of Apple Computer.

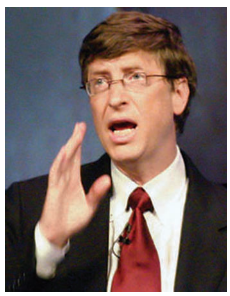


Figure 1.19
Bill Gates, cofounder of Microsoft Corporation.

(a) (b)

Figure 1.20
Vinton Cerf and Robert Kahn, inventors of the Internet.

Figure 1.21
Donald Knuth, father of the analysis of algorithms.

Figure 1.22
Tim Berners-Lee, inventor of the World Wide Web.

Internet is based. Vinton Cerf created the first commercial Internet e-mail system and is now Vice President and Chief Internet Evangelist for Google.

Donald Knuth (Figure 1.21), a computer scientist, mathematician, and Professor Emeritus at Stanford University, has been called the father of the analysis of algorithms. His multivolume set of books entitled *The Art of Computer Programming* is the classical reference for all programmers. He is also the developer of the text document (TEX) typesetting system for creating high-quality digital publications.

Tim Berners-Lee (Figure 1.22) is known as the inventor of the World Wide Web and continues to direct the Web's development as the director of The World Wide Web Consortium (W3C). He is also a director of the World Wide Web Foundation, which furthers the potential of the Web to benefit humanity.

1.2.5 Smaller, Faster, Cheaper Computers[4]

Computing has made more progress in 15 years than transportation has made in 2,000 years, having gotten smaller, faster, and cheaper during that time. Your cell phone today is about a million times cheaper, a thousand times more powerful, and a hundred thousand times smaller than the one computer that was used at MIT in 1965.

According to Ed Lazowska, chairman of the University of Washington's Computer Science and Engineering Department, if Detroit car makers could have paralleled the innovations that hardware and software manufacturers have realized for computers, today's cars would be tiny, powerful, and inexpensive. They would be as small as toasters, cost $200, travel 100,000 miles per hour, and would run 150,000 miles on a gallon of fuel. "In Roman times, people traveled along on horses or in carts at about 20 miles per day," he said. "In the early part of this century, the automobile allowed people to travel at 20 miles per hour. Today, supersonic military aircraft travel at about 20 miles per minute. That progress is about a factor of 1,000 in about 2,000 years," Lazowska wrote in an e-mail message.

Another analogy by Rick Decker and Stuart Hirshfield in *The Analytical Engine* states, "If automotive technology had progressed as fast as computer technology between 1960 and today, the car today would have an engine less than a tenth of an inch across, would get 120,000 miles per gallon, have a top speed of 240,000 miles per hour, and would cost $4.00."[5] Also, at a recent Computer Dealers Exhibition (COMDEX) meeting, Bill Gates is reported to have said that if GM had kept up with technology like the computer industry has, we would all be driving $25 cars that get 1,000 miles per gallon.

Computers and the programs that provide their instructions will continue to increase in speed, reliability, and functionality, limited only by human creativity.

1.3 SPECIFYING A PROGRAM

As discussed in Section 1.1, an increasingly large number of people own and use a computer as part of their everyday lives, yet a very low percentage of these computer users actually know how to write a computer program. In fact, if you understand the material in the first two sections of this textbook, you already know more about computer programming than most of the world's population. As a result, most programs are not written by the program users. Rather, they are written by a group of computer processionals most people would refer to as programmers, but more accurately, they should be called **software engineers**. A new program that does not meet the needs of the end user is not going to be well received, so it is important that there be a way to describe the requirements of a new program in a way that is understandable to the end users.

Definition

A **software engineer** is a computer professional who produces programs that are on time, within budget, are fault free, and satisfy the end users' needs.

The more formal techniques for describing the requirements of a new program are part of the discipline of systems analysis, which is a subset of software engineering. These formal techniques are all based on one specification of the arrangement of the components of the computer system shown Figure 1.4. They assume that the users' interaction with the program is via the input and output devices, so the simplest way for end users to define what task the program is to perform is to enter into conversation with a systems analyst aimed at defining the inputs to, and the outputs from, the program.

For example, suppose your friend Annie recently purchased a computer and is having trouble managing her money. Knowing you completed a course in computer programming, she comes to you for help. You and Annie enter into the following conversation, which typically involves the probative words who, what, why, where, when, and how:

Annie: I want to know *where* my money goes.

You: OK Annie, *what* bills do you pay each month?

Annie: Well, there's food, rent, electric, telephone, and clothing.

You: *How* much is each bill?

Annie: That's part of the problem; they change each month, and so does my income because I work on commission.

You: Well, do you know roughly *what* percent of your income is spent on each?

Annie: No, but I sure would like to know that. I have a feeling some months I'm spending too much of my income on food and clothing, which leaves me with no mad money.

You: *What* is mad money?

Annie: You know, money I can spend on anything I like other than these bills. I want to know *how* much that is each month. I am sure someone is taking my money.

You: Gee, Annie, you sound a little paranoid.

Annie: You'd be paranoid too if everyone was out to get you!

You (whispered): Why do I bother?

Based on this conversation, you know the two things Annie would like her computer system to determine and output are the amount of "mad money" (discretionary funds) she will have at the end of a month and the percent of her monthly income she spent on each of her five monthly bills. To determine this, she will have to input the amount of each of her five monthly bills and her income for that month. You have decided to include the month and year as two additional inputs to the program, so she will be able to save and distinguish one month's results from another. A simple description, or specification, for this program is shown below. It is a tabulation of the program inputs and outputs preceded by the name of the program and a brief statement that describes the overall task the program performs.

Program Specification

Program Name: *Annie's Money Manager*

Task:	To determine Annie's monthly discretionary funds and the distribution of her monthly expenses
Inputs (8):	Month and Year Income for the month Amount spent during the month on each of the following five items: food, rent, electric, telephone, and clothing
Outputs (6):	Percent of monthly income spent on each of these five items: food, rent, electric, telephone and clothing Amount of discretionary funds

Typically, the program specification is refined through an iterative process that involves its review by the end user and a subsequent conversation. This process could introduce more functionality into the specification of Annie's program. For example, it could also include the ability to output an annual report showing the values of the six outputs for any given year, or perhaps for a range of months. Obviously, this would expand the specification given above.

Given the specification of the program, the programmers' goal is to write a program that accepts the specified inputs and produces the desired outputs. The programmers may have to consult with other experts if it is unclear to them how to determine the outputs from the given inputs. For example, if the programmers assigned to write Annie's program did not know how to compute percentages, they would have to consult a mathematician.

1.3.1 Specifying a Game Program

The technique discussed to specify Annie's program is similar to that of specifying any program: conduct a brief conversation with the user and then tabulate the program's name, the task it performs, the inputs, and the outputs. This approach can also be used to specify a program that is used to play a game. In addition, the realization that all game programs share a common set of features can facilitate the specification process if the systems analysts include questions about these features in the conversations they have with the game's inventor.

For example, most games involve game objects (e.g., trucks, cars, and a frog). In addition, all games have an objective or a way to win the game (e.g., moving a frog object to the other side of a road without having it run over by a truck or a car). Most games also have other features in common. A list of common features to include in a game's specification conversation is given in Figure 1.23.

- Name of the game
- Objects (starships, trucks, sling shots, etc.) that will be part of the game
- Objective of the game
- Way to calculate the score of the game
- Time limits imposed on the game
- Game pieces (objects) that will be animated
- Game pieces controlled by the game player (the program's user)
- Input devices used to control the game objects
- Particular colors to include in the game
- Determining when the game ends
- Events that take place when the game ends
- Keeping track of the highest game score achieved and the name of the game player who achieved it

Figure 1.23
Common game features.

Armed with this checklist of common game features, a typical conversation with your friend Ryan (an aspiring game inventor who has not taken a programming course) could be:

You: Hi Ryan, what's up?

Ryan: I've got a great idea for a video game called Deep Space Delivery.

You: *What* is the objective of the game?

Ryan: To deliver as many supply packets as possible (picked up from a supply depot) to five different planets before time runs out.

You: *How* is a player's score calculated?

Ryan: The player gets one point for each packet delivered, and if the player delivers all of the packets at the depot before the time runs out, the player receives one point for each second of time remaining.

You: *What* is the time limit on the game, and *how* many packets will be in the depot?

Ryan: One minute and 30 packets.

You: Looks like the game pieces (objects) are the planets, the supply packets, and the supply ship. Is that correct?

Ryan: Yes, but don't forget to include the supply depot.

You: *How* will the player move the supply ship and pick up and drop off the packets?

Ryan: Using keys on the keyboard.

You: Will any of the other game pieces be moving?

Ryan: Yes, the planets will be moving and bouncing off the edges of the game board. Also, make one of the planets white and another red.

You: Would you want to keep track of the highest game score achieved and the name of the game player that achieved it?

Ryan: Yes, that's a good idea.

You: Sounds good Ryan. I'll write up a specification for the game for you to look over. Then, I'll write the program, and we'll split the profits. How's that sound?

Ryan: How about a 40% share for you?

Ryan (whispered): It's all my idea.

You: OK.

You (whispered): But I'm doing all the work.

Based on this conversation, the specification of Ryan's game is given below.

Program Specification

Program Name: *Deep Space Delivery*

Task:	A starship is to pick up supply packets at a supply depot and deliver as many supply packets as possible to five moving planets before time runs out. The player will receive one point per packet delivered and one point per second remaining on the game time after all packets are delivered.
Inputs (7):	The four cursor control keys (up, down, right, and left) used to control the position of the starship

	The 'A' key, which is used to pick up a supply packet when the ship is at the supply depot
	The 'Z' key, which is used to drop off a packet when the starship is at a planet
	The game player's name input when the game is launched
Outputs (5):	The time remaining in the game, in seconds, beginning from 60 seconds
	The player's score
	The message "Game Over" when the game time reaches zero, or when all packets are delivered
	The highest score achieved and the name of the person who achieved it to be output to the game board and a disk file when the game time reaches zero

The details for more functionality could be added to the specification of Ryan's program. For example, the delivery of a packet to a faster moving planet could be awarded multiple points, multiple levels of difficulty could be added to the game, and the highest game score achieved with the name of the game player who achieved it could be announced at the beginning of the game.

1.4 SAMPLE STUDENT GAMES

We will soon be able to write a program that implements the specification of the Deep Space Delivery game presented in the preceding section. The game programs on the DVD that accompanies this textbook were specified and written by students enrolled in an introductory programming course. To run these programs, simply double click the "Sample Student Games" folder on the DVD and copy the subfolders onto your hard drive. Double click one of the subfolders and then double click the file with the .jar extension. Running these programs will give you a sense of what you will be able to accomplish after gaining an understanding of the material in the first five chapters of this book.

1.5 JAVA AND PLATFORM INDEPENDENCE

A computer system's platform is the CPU model and the operating system software it is running. For example, many PCs run on an Intel CPU/Windows platform, and Apple computers manufactured after the midpoint of 2011 run on an Intel CPU/OS X platform. As a result of the evolution of CPUs and operating system software that has taken place over the last 30 years, there are many different platforms in use today.

The variety of platforms has always been a problem for software developers because each platform has a language of its own, meaning that a platform can only execute a program that is written in its language. To produce a program that could run on two different platforms, the programmer either had to write the program twice, first in the language of one platform and then in the language of the other, or write the program in a more generic language (for example, C++) and then use two other programs to translate that program into the language of the individual platforms. In this case, the C++ program is referred to as source code, and the resulting translations of this source code are called executable modules or executables.

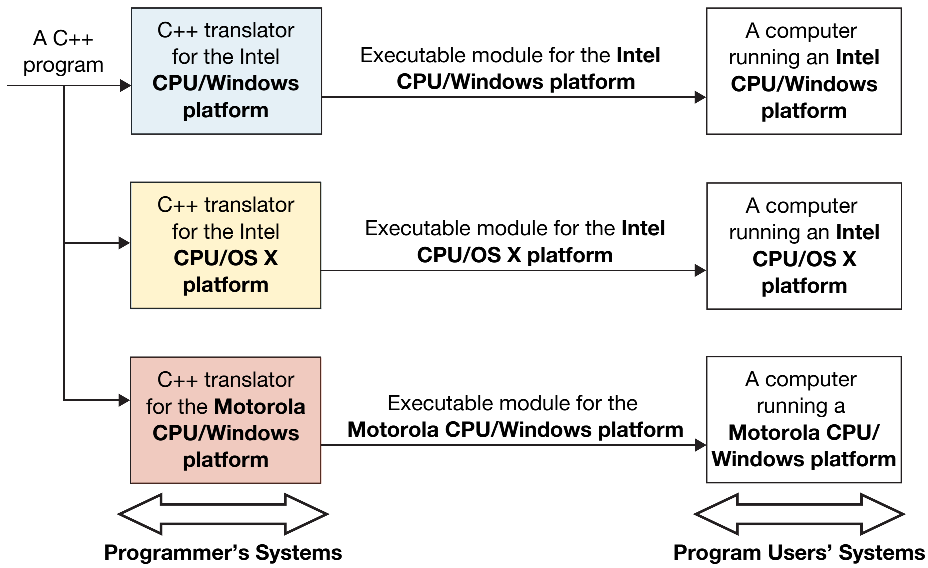


Figure 1.24
The C++ multiple-platform translation process.

When we consider the number of platforms that exist and the fact that writing in the language of a particular platform is a very tedious and time-consuming process, writing programs in a generic language is the most efficient and cost-effective approach. Figure 1.24 illustrates the use of this process to produce executable modules for three different platforms. The programmer would have to translate the program using three different translators to generate the three different executable modules.

During the early 1990s, the Internet was made commercially available to private individuals, which made it possible for them to share information between their computer systems. The idea that this information could be a program resident on one computer system (perhaps a program to display a Website) presented a fundamental problem. If the two computers were not running the same platform, the executable module downloaded from one platform (the host platform) would not run on the other (the client platform), and the Website would not be displayed on the client machine. Using the process illustrated in Figure 1.24 to produce a downloadable executable module for all platforms was an impractical solution because, for one thing, a program that was written today should be able to be run on the platforms of tomorrow. Fortunately, a team of computer scientists at Sun Microsystems lead by James Gosling had already come up with a more practical solution.

The team's idea was to change the process used to produce an executable module. Instead of the host machine producing the executable module, the client machine would produce it. The host machine would simply translate the program, written in a new programming language named Java, into a set of byte codes. Byte codes should be thought of as a pseudo-executable module for a virtual machine, named the Java Virtual Machine (JVM), which are not in the language of any platform in existence. Once generated, the byte codes could then be downloaded to any client machine, and the client machine would use a byte code translator program to translate the downloaded byte codes into the language of its platform. Figure 1.25 illustrates the Gosling team's new process.

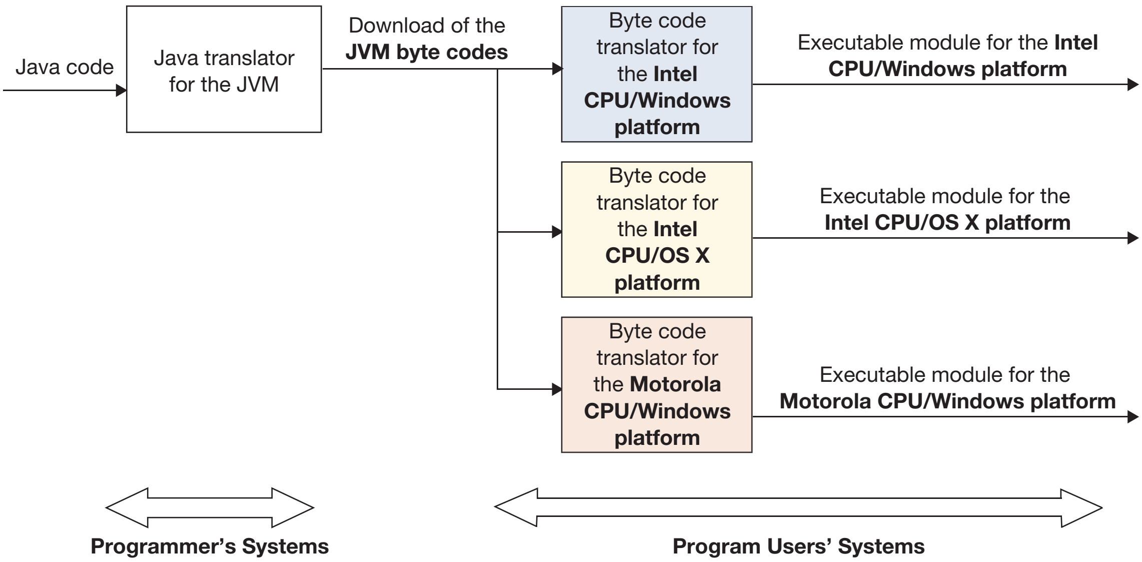


Figure 1.25
The Java multiple-platform translation process.

NOTE *The grammatical rules for writing a program in the Java language were described by the Gosling team in the Java Language Specification (JLS), available online.*

To make this process work, Gosling's team assumed that the manufacturer of the client computer system would install a translator that translated Java byte codes into the language of their system's platform. Realizing that all future customers would want to attach their new computer to the Internet, computer manufacturers complied and proudly advertised their system as "Internet ready."

The fact that the same set of byte codes could be downloaded and used to produce an executable module on any platform that had a byte code translation program on it made Java programs platform independent. Programs written in Java (or more accurately, the program's byte codes) could be downloaded, translated, and then executed on any platform that contained a platform-specific byte code translator.

1.5.1 The Java Application Programmer Interface

In addition to providing a translator that translates Java programs into byte codes, the creators of Java also identified a group of data (e.g., the mathematical constant pi) and tasks (e.g., computing the square root of a given number) that were likely to be used in Java programs. A description of these data and tasks was then published as the Java Application Programming Interface (API) specification. If a Java programmer wanted to create a new window for a program, which normally most programmers want to do, it could be easily done by incorporating the API task that contained all of the Java byte codes necessary to display a window into that program.

For ease of use, the data and tasks that are similar were grouped together. These grouping are called packages, and within the packages there are subgroupings called **classes**. There are approximately 200 packages and 4,000 classes in the Java API. Most of these classes contain both data and the Java instructions to perform common tasks. A set of instructions to perform a task is called a method. The data and methods that are in the same class are said to be **members** of the class.

Definition

A **class** is made up of a group of related data members and member methods.

A **method** is a set of instructions used to perform a task.

Data members are the instance variables that contain the data values for the class.

Just as Java's creators assumed that the manufacturers of computer systems would install a translator that translated byte codes into the language of their system's platform, they also assumed that the manufacturers would install an implementation of the data and methods defined in the API specification. Once again, to advertise that their system was Internet ready, the manufacturers complied. Technically speaking, the byte code translator and the API implementation on the client machine (along with a memory manager) are called the Java Runtime Environment (JRE), and the JRE and the client system's operating system are considered an implementation of the Java Virtual Machine. Figure 1.26 gives the components of the Java Virtual Machine specific to a system running an Intel CPU/Windows platform.

Based on Gosling's team's idea, any programming language can achieve platform independence if the language designers provide a translator that translates the language into Java byte codes. The resulting translation will run on any computer system or mobile device that implements the Java Virtual Machine.

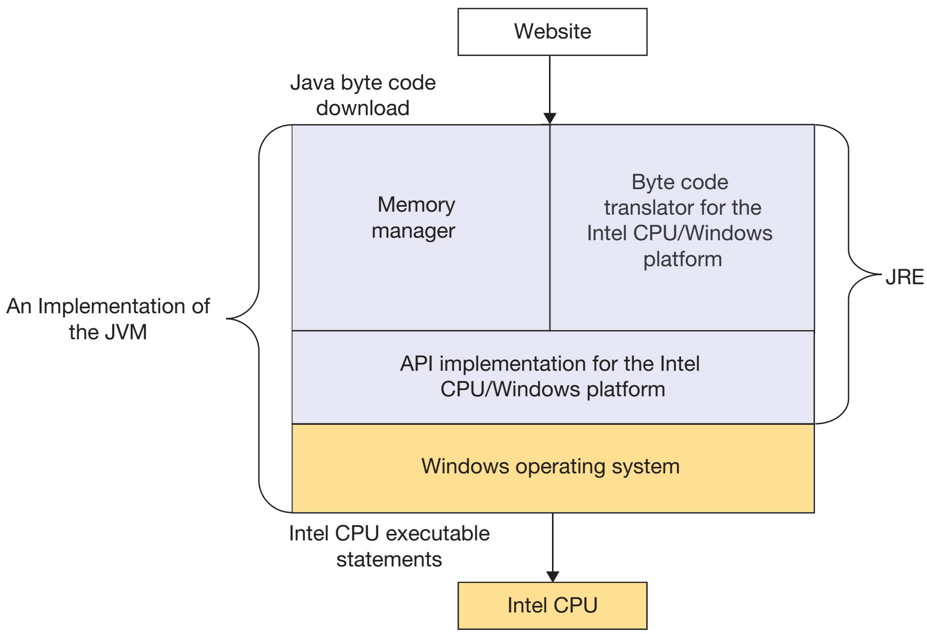


Figure 1.26
System-specific components of the Java Virtual Machine.

1.6 OBJECT-ORIENTED PROGRAMMING LANGUAGES

Just as related methods and data are grouped into classes in the API specification, they can also be grouped into classes that are defined within programs written using object-oriented programming (OOP) languages. Java, by design, is an OOP language. Grouping related methods and data inside a class that is defined in a Java program is more than a convenient way of arranging related data and methods. The real motivation for permitting this class grouping in object-oriented programming languages is that it is a good way of modeling the objects that the program will deal with.

As an example, consider a video game program that involves starship objects. Each starship object will have a name and a (x, y) location. In addition, as the game is played a new starship can be created, starships can be drawn on the monitor, and a starship's location can be changed. A good model for these starship objects would be to define a class named Starship (depicted as the blue rectangle in Figure 1.27). As shown in the figure, the class would have three data members (name, x, and y), and three member methods (create, draw, and move).

It is important to understand that a class is not itself an object, but rather it is a description of an object. From one class we can create an unlimited number of **objects** or instances of the class. A useful analogy is to consider classes we encounter in everyday life: a blueprint, a cookie cutter, a stencil, a pottery mold, a dress pattern, and the human genome pattern. From one blueprint we can create lots of houses, from one cookie cutter lots of cookies, from one stencil lots of pictures,

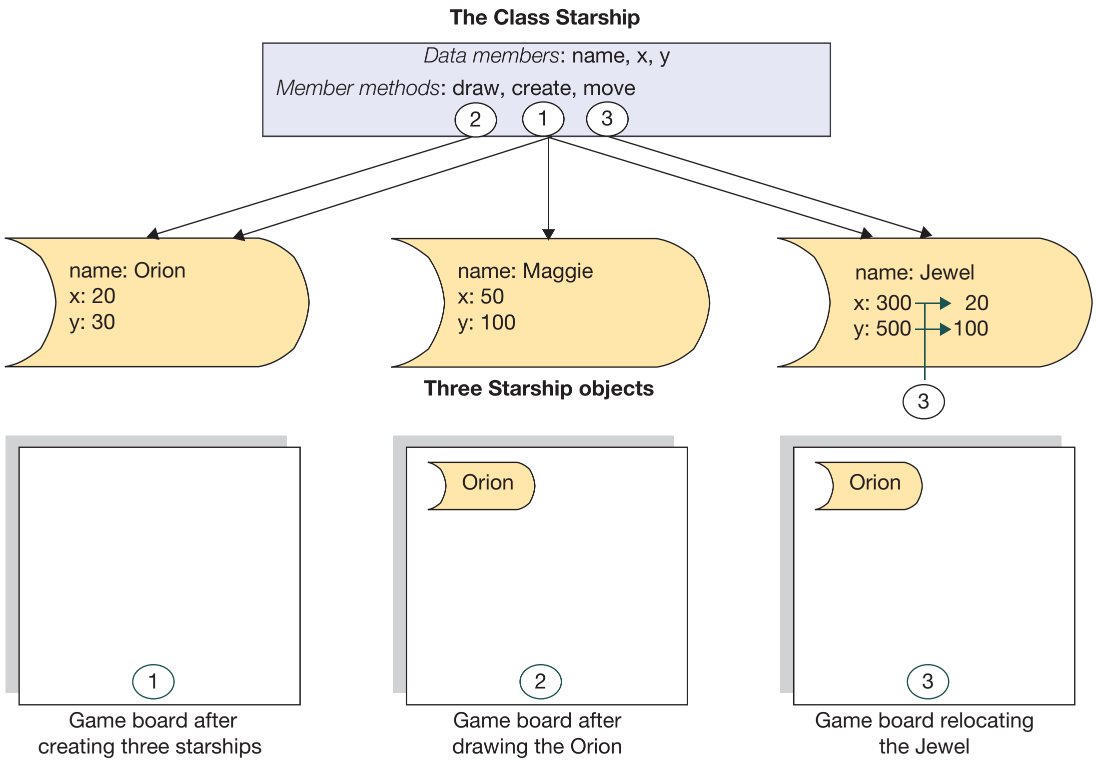


Figure 1.27
The Starship class, three Starship objects, and the use of the class's methods.

from one pottery mold lots of vases, from one pattern lots of dresses, and from one human genome pattern lots of people.

Definition

In an object-oriented programming language, a **class** is a template for an object, and an **object** is a particular instance of a class.

The Starship class would be a template for a starship object. Each time a starship enters the game, a new starship would be created from this template with a given name and initial (x, y) location using the class's create method. A starship's name and (x, y) location would be stored in its three data members, which each object created from the class Starship would contain. In addition, the tasks of drawing and relocating a starship would be performed by the Java instructions that make up the class's draw and move methods.

The center and bottom sections of Figure 1.27 depict the use of the Starship class's three member methods used in the following order:

1. The create method (indicated by the number 1 in the figure) was used to create or construct the three starship objects shown in the center of the figure: the Orion at (20, 30), the Maggie at (50, 100), and the Jewel at (300, 500). Notice that after they are created, each starship contains three data members to store the ship's name and its (x, y) location. Although these three starships have been created, they are not displayed on the game board shown at the lower left portion of the figure because the draw method has not been used.
2. The draw method (shown as number 2 in the figure) was used to display the starship Orion at its current location (20, 30), as depicted in the bottom center of the figure. The draw method has not operated on the other two starships, so, even though they exist, they do not appear on the game board. (Note: The origin is located at the upper left corner of the game board and positive y is downward.)
3. The move method (represented by the number 3 in the figure) was used to change the current location of the starship Jewel from (300, 500) to (20, 100) as depicted on the center right portion of the figure. As shown at the bottom right portion of the figure, it is not displayed because the draw method was not performed on it. After relocating the starship, if the draw task were performed on the Jewel, it would have been displayed directly below the Orion at (20, 100).

NOTE *Each object contains the data members of its class and can be operated on by the class's methods.*

1.7 INTEGRATED DEVELOPMENT ENVIRONMENTS AND THE PROGRAM DEVELOPMENT PROCESS

An Integrated Development Environment (IDE) is a program to help programmers write programs. Usually they are language specific in that a particular IDE can be used to develop programs in one, and only one, programming language. For example, NetBeans and Eclipse are two popular

IDEs used to develop programs written in Java, and the IDE Microsoft Visual C++ can be used to develop programs written in the language C++. Many popular IDEs can be downloaded for free from the IDE's Website.

What these programs have in common is that they integrate a set of program development tools into one program. Examples of these tools are a text editor used to type, edit, save, and re-open the program's instructions, and a translator used to translate the program instructions into the language of the platform it is to run on. In the case of a Java IDE, this would be a translation from Java into Java byte codes. In addition, most IDEs have an autocomplete feature to facilitate the typing of the program and a grammar checker to help locate and correct grammatical errors in the program's instructions.

Armed with a good specification of a program and a good IDE, we are almost ready to begin the program development process, which is illustrated in Figure 1.28. Before we begin, we must read the program's task contained in its specification and discover a set of algorithms that perform the tasks. For example, how will we determine when a starship delivers a supply packet to a planet in Ryan's Deep Space Delivery program? As mentioned at the end of Section 1.1, this can be the most difficult part of writing a program, and most software engineers take an advanced course in algorithm discovery. We will illustrate the discovery process via the programming examples presented throughout this textbook.

After discovering the program's algorithms, we are ready to begin the program development process (Figure 1.28). Generally, the process begins with representing the algorithms as a set of program instructions (called code), translating the code, and then correcting the grammatical errors (called syntax errors in computer science). Once all of the syntax errors have been eliminated, the IDE's translator will produce an executable module that it then runs. In the case of Java, the IDE generates and then executes the Java byte codes on the Java Virtual Machine installed on the programmer's computer.

The programmer then changes roles from programmer to program user to test the program for correctness. To do this, the user (or tester) supplies the inputs to the program and examines the out-

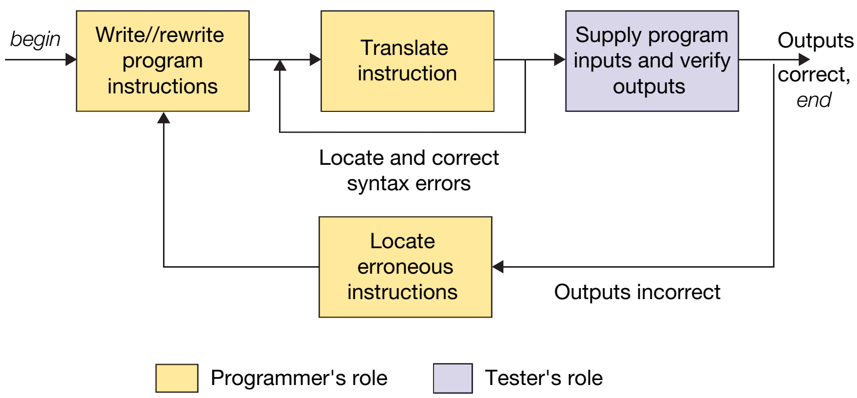


Figure 1.28
An overview of the program development process.

puts it produces. If the program produces the correct outputs for several well-chosen sets of inputs, the program is complete. If it does not, the tester changes back to the role of a programmer, locates the erroneous instruction(s), and the process is repeated beginning with rewriting those instructions.

One refinement to the process is necessary for anything other than a very, very small program because of the fact that we cannot effectively solve large problems. When we consider that we humans have visited the moon and that many of the more common operating systems consist of over a million lines of instructions, this statement leads us to a paradox: If we can't solve large problems, how did we do these things?

The answer lies in the 4,000 BC writings on a Chinese cave wall that explain that big things can be divided into little things, little things can be divided into nothing. Today's version of this is: divide and conquer. Just as the task of going to the moon was divided into hundreds of small problems whose solutions were integrated into the lunar mission, a large program is divided into many small parts, which can be combined to become the large program.

Object-oriented programming languages present several obvious dividing lines. Because the specification identifies the types of objects the program will deal with, the program is first divided into classes, one for each type of object. Then, within each class, the tasks to be performed on the class's objects are defined. Simple tasks become member methods, complex tasks are divided into several simple tasks (each of which also becomes a member method). Each method within a class is written and tested separately. Basically, each method is considered to be a small program, and it is developed using the process illustrated in Figure 1.28. Once all of the methods in a class are operating correctly, the methods in another class are developed using this divide-and-conquer concept. When all the classes are complete, they are integrated into the large program. Object-oriented programming languages make it easy to integrate the classes into the large program.

As an example, consider the development of the Starship class shown in Figure 1.27, which is part of a game program. Three methods (create, draw, and move) have to be developed using the process illustrated in Figure 1.28. Because we cannot draw or move a Starship that has not yet been created, the create method would be developed first. After the method is written and the syntax errors are found and corrected, we would write a few more lines of Java to test the method. This code is often referred to as **driver** code because it takes the method for a "test drive." It would use the method to construct a Starship object, perhaps Maggie in the center of Figure 1.27, and output its data members. If the name Maggie and position (50, 100) were output, we would conclude the create method was working. If not, we would examine the instructions that make up the create method, locate and correct the errors, and repeat the translation and test portion of the process.

NOTE *Driver code is disposable Java code used to test a method. It normally does not become part of the final program's instructions.*

The next logical step would be to develop the draw method because, as we will see, it can be used in the testing of the move method. After the syntax errors are found and corrected, we would write a few more lines of Java driver code to test the method. The code would use the create method to construct a Starship object, perhaps Orion shown on the left side of Figure 1.27, and then use the draw method to display it on the game board. If it were displayed in its proper location with the name Orion on the side of the ship, we would conclude the draw method was working. If not, we would examine the instructions that make up the draw method, locate and correct the errors, and then repeat the translation and test portion of the process.

Next, we would develop the move method, write its code, translate the code and correct the syntax errors, and then write a few more lines of Java driver code to test it. The code would create a Starship object, perhaps Jewel at (300, 500), as shown on the right side of Figure 1.27, then use the move method to change its position to (20, 100) and the draw method to display it on the game board (monitor). If it were displayed in its new location (20, 100) with the name Jewel on the side of the ship, we would conclude the move method was working. If not, we would examine the instructions that make up the move method, locate and correct the errors, and repeat the translation and test portion of the process.

After completing the development of our Starship class in three manageable steps, we would eliminate the driver code and replace it with the instructions to use the Starship class and its methods in our game program.

1.7.1 Mobile-Device Application Development Environments

The level of miniaturization of the basic components of a computing system that has taken place in the last ten years has brought to the marketplace a variety of hand-held computing devices. These devices, often referred to as mobile devices, include smart phones, personal digital assistants (PDAs), and tablet devices.

The development of a program for a mobile device follows the same process as that used to develop a program for a non-mobile computing device discussed in this chapter. After a specification is written and the program's algorithms are discovered, an IDE is used to develop the specification into a functional program using the process shown in Figure 1.28. However, two problems arise when applying the development process to mobile-device applications. Because these devices have limited computing power, it is impractical to conduct the development process on them, so the process is conducted on a more powerful non-mobile computing system. In addition, the concept of platform independence has not been extended to mobile devices, so an executable module must be produced for each mobile-device platform.

Because a majority of mobile applications run on smartphones, and a great majority of smartphones run an Android-based platform, this section will conclude with an overview of the tools available for developing applications for any Android-based smartphone or tablet device. Although the details presented are specific to those devices, the concepts presented are typical of the tools employed to develop applications on most mobile devices.

Android device applications can be written in Java on a personal computer. The preferred IDE is Eclipse, which is a free download. Eclipse is preferred because two sets of tools that facilitate the development of an Android-device application are easily integrated into it. Both of these tools can be freely downloaded. The first of these, the Android Software Development Kit (SDK), can be downloaded from the Android developers' Website. The second set of tools, the Android Development Tools (ADT) Eclipse plug-in can be downloaded from the Eclipse Website. If your personal computer is running a Windows operating system, you can download the Eclipse IDE, the SDK, and the ADT as one bundle from the Android Developers Website, found at *http://developer.android.com/sdk/index.html*.

Some of the features the two sets of tools provide include:

- The latest version of the Android operating system
- Platform-dependent translators
- A set of emulators that run the translated code on a simulation of any Android-based mobile device including displaying its screen and emulating all of its I/O functionality
- The ability to upload developed applications to the Android Market (a Web-based store for free and purchased applications)

Using these tools and knowledge of Java, you will be able to develop and market applications for any Android device from the comfort of your own personal-computer system.

1.8 OUR GAME DEVELOPMENT ENVIRONMENT: A FIRST LOOK

In Section 1.4, you were asked to run several of the sample game programs contained on the DVD that accompanies this textbook. The DVD also contains a folder named Package that contains a Java package named `edu.sjcny.gpv1`. This package can be thought of as a game development addition to the API because it contains methods that perform tasks that are common to most game programs. Appendix A contains descriptions of the methods contained in this package.

The incorporation of this package, or game development environment, into a game program facilitates its development. The students who created all the sample game programs contained on the book's DVD incorporated it into their programs. In this section, we will describe how to easily create and display a game window using the methods in this package, how to incorporate the package into a game program, and how to change some of the game window's properties.

1.8.1 The Game Window

When incorporated into a Java program, two of the game environment's methods can be used to create and display the game window shown in Figure 1.29. The Pause and Start buttons on the right side of the game window can be used by the game player to pause the game and to start/restart the game. The directional buttons below them, or the keyboard keys, can be used to control the position of the game objects during the game.

The coral-colored area on left side of the game window, called the game board, is where the game objects appear. Like most windows, it can be dragged around by its title bar, minimized to the status bar, and redisplayed by clicking its icon on the status bar. It cannot be maximized, however, the programmer can change its size to accommodate the needs of a particular game. The default size of the game window is 622x535 pixels, which are closely spaced dots of color that make up the surface of a computer monitor.

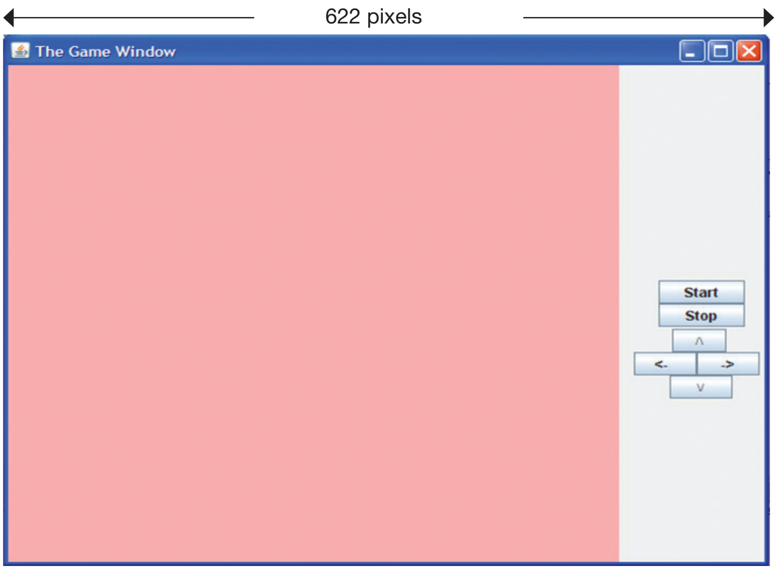


Figure 1.29
The game environment's window.

1.8.2 The Game Board Coordinate System

Figure 1.30 shows the game board coordinate system. Game objects are positioned on the game board by specifying their x and y game board coordinates. The system is a two-dimensional Cartesian system with its origin at the upper left corner of the game window. The positive x direction is to the right, and positive y direction is down. The units of the axis system are pixels.

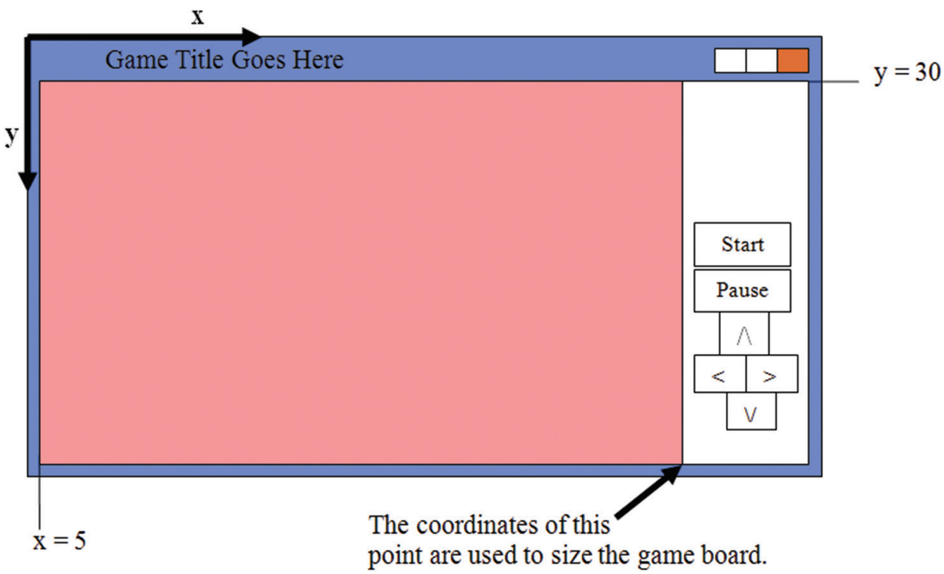


Figure 1.30
The game board coordinate system.

As shown on the upper right and lower left sides of Figure 1.30, the title bar of the window is 30 pixels high, and the left boundary of the window is 5 pixels wide. The coordinates of the lower right corner of the game board for the default window size are (500, 500). If the programmer decides that a larger or smaller game board is appropriate for the game being developed, the x and y coordinates of the lower right corner of the game board can be changed, which will be described in Section 1.8.5.

1.8.3 Installing and Incorporating the Game Package into a Program

on the dvd

Appendix B contains detailed instructions on how to incorporate the game package, which contains the game development environment, into a Java program. The simplest approach is to use one of the projects contained in the "IDE Specific Tools" subfolder on the DVD that accompanies this book. This subfolder contains an Eclipse, a NetBeans, and a JCreator project that has the game package already incorporated into them as well as the code described in the next section, which creates and displays the game window. When the projects are run, they display the game window shown in Figure 1.29. Game program specific code and classes can be added to them.

The JCreator and NetBeans projects on the DVD can be copied from the DVD and pasted into a folder, and then the project can be opened, modified, and run from within the IDEs. The Eclipse project must be imported into an Eclipse workspace folder using the Import feature available on the Eclipse File drop-down menu. After the Eclipse project is imported from the DVD, it can be opened, modified, and run from within Eclipse. Detailed instructions on the use of the DVD's three preexisting game projects are given in Appendix B.

As an alternative, the game package edu.sjcny.gpv1 in the "Game Environment" folder on the DVD can be added to any newly created Java project by following the procedures given in Appendix B, most of which do not include having to change the system's CLASSPATH variable. When these alternative approaches are used, the code described in the next section, which creates and displays the game window, must be added to the project's code.

1.8.4 Creating and Displaying a Game Window and Its Title

After you have incorporated (imported) the game package into your program, you can use the methods in the package to create and display the graphical window in which your game will run. The Java program shown in Figure 1.31 is a template, or starting point, for all of our graphical game application programs. When this program is run, the game window shown in Figure 1.29 is created and displayed.

As we will learn in Chapter 2, lines 2, 3, 7, 8, 10, and 11 are the minimum set of instructions that make up a Java application program. For that reason, many IDEs generate these instructions when a new programming application is created. The one exception is the phrase **`extends`** `DrawableAdapter`, which must be added to the end of line 2 if the game package is to be used in the program. Lines 1, 4, 5, and 9 complete the game program template.

```
1  import edu.sjcny.gpv1.*;
2  public class GameWindowDemo extends DrawableAdapter
3  {
4     static GameWindowDemo ga = new GameWindowDemo( );
5     static GameBoard gb = new GameBoard(ga, "The Game Window");
6
7     public static void main(String[] args)
8     {
9        showGameBoard(gb);
10    }
11 }
```

Figure 1.31
The Java instructions to create and display the game window.

The import statement on line 1 of Figure 1.31 makes the methods in the game package available to the program. Lines 4, 5, and 9 use these methods to create and display the game window shown in Figure 1.29. Each Java program is given a name, which is part of its specification. This program is named `GameWindowDemo`, which is typed on line 2 after the word `class` and typed two more times on line 4.

As previously mentioned, Figure 1.31 will be the template for all of our graphical game application programs. To adapt it to a particular game program, the new program's name would appear on lines 2 and 4, and the game's title and perhaps the name of its creator would appear at the end of line 5. For example, if a new game program's name was Project1, and the game was Frogger created by Bob, the changes to lines 2, 4, and 5 would be as highlighted below:

```
1  import edu.sjcny.gpv1.*;
2  public class Project1 extends DrawableAdapter
3  {
4    static Project1 ga = new Project1 ( );
5    static GameBoard gb = new GameBoard(ga, "Frogger, by Bob");
6
7    public static void main(String[] args)
8    {
9       showGameBoard(gb);
10   }
11 }
```

1.8.5 Changing the Game Board's Size

As mentioned in Section 1.8.1, the default size of the game window is 622x535 pixels. This was chosen to make the coordinates of the game board's lower right corner (500, 500). To change the game board's size, and thus the window size, we add the new coordinates of the game board's lower right corner to the end of line 5 of Figure 1.31. This is the line that constructs the window. For example, to obtain a game board whose lower right corner is located at (700, 650), we would change line 5 to:

```
static GameBoard gb = new GameBoard(ga, "The Game Window",700, 650);
```

The title bar of the window would still be 30 pixels high, and the left border of the window would still be 5 pixels wide, as shown in Figure 1.29, but the window's height and width would be increased to accommodate the larger game board.

1.9 REPRESENTING INFORMATION IN MEMORY

As discussed in Section 1.1, the memory component of the computer system has the ability to store and recall information, and that information could be the data that the program processes or the instructions that make up the program. The scheme used to store or represent the information in memory is dependent on the type of information being stored. Data is stored using a different scheme than translated program instructions. In addition, character data, which is data typed into a word processor or IDE, is stored differently than numeric data, which is data that will be used in arithmetic expressions.

There are three memory storage schemes used to represent three different types of information: (1) character data, (2) translated instructions, and (3) numeric data. All three of these schemes were designed around the basic hardware memory unit: a bit, which stands for *b*inary dig*it*. Conceptually, a bit should be thought of as a single switch that can be turned on or off. All of memory uses this storage concept, and storage devices such as RAM, disks, flash drives, and tape drives may contain billions (giga) and even trillions (tera) of these bits.

For brevity, when a bit is turned on we say it is in state 1 (one), and when it is off we say it is in state 0 (zero). These should only be thought of as the numerics one and zero when the information stored is numeric data. Figure 1.32 depicts eight adjacent bits in on-off states and their briefer binary (1-0) depiction.

off on off off off off on off	01000010
on-off Depiction	**1-0 Depiction**

Figure 1.32
The state of eight adjacent on-off bits and their 1-0 depiction.

1.9.1 Representing Character Data

The scheme used to represent character data in memory is rather straightforward. A table[7] was composed in which each character to be represented was assigned a unique eight-bit pattern. For example, the character B was assigned the pattern 01000010, the lower-case version of this character, b, was assigned the pattern 01100010, and the character 1 was assigned the pattern 00110001.

The table is named the Extended American Standard Code for Information Interchange because it was an expansion of a table named the American Standard Code for Information Interchange, which represented characters using patterns of seven bits. The seven-bit table was assigned the acronym ASCII (pronounced "ask ee"), and the extended table is referred to as the Extended ASCII table. Both tables include all of the upper- and lower-case letters of the Modern

Latin (English) alphabet, the digits 0 to 9, a set of special characters (e.g., !, @, #, $, %, ^, etc.), and some control characters such as horizontal tab and line feed. Because there are 128 (2^7) unique ways to arrange 7 bits and 256 (2^8) unique ways to arrange 8 bits, adding the eighth bit to the Extended ASCII table doubled the size of the ASCII table.

The first 128 characters in the Extended ASCII table are given in Appendix C. The bit patterns in this table are used to represent character information on all computer systems when the alphabetic characters the system is processing are limited to the Modern Latin (English) alphabet. When this is the case, and we want to represent the letter B in storage, eight adjacent (or contiguous) bits of storage (called a **byte** of storage) are set to the Extended ASCII pattern for B: 01000010. If we fetched a byte of storage from an area of memory in which we knew that characters were stored, and that byte contained the pattern 01000010, we would know that the character B was stored there. We say that a keyboard is an ASCII keyboard if it generates this bit pattern when a capital B is struck, and a printer is an ASCII printer if it prints the character B when it receives this bit pattern.

Definition
Eight adjacent or contiguous bits are called a **byte** of storage

To accommodate the international exchange of information over the Internet, the Extended ASCII table was expanded to include unique bit patterns for the symbols used in the other alphabets of the world. To provide a unique bit pattern for each entry in this expanded table, named the UNICODE table, the number of bits assigned to each character was increased from 8 to 16 bits (2 bytes) per character. The first 256 entries in the UNICODE table are the characters in the Extended ASCII table, with the leftmost 8 bits of their 16-bit pattern set to 0 and the rightmost 8 bits set to their Extended ASCII table patterns. For example, because the Extended ASCII representation of B is 01000010, its UNICODE representation is 00000000 01000010. Characters processed by Java programs are stored in memory using their UNICODE table representations.

NOTE *Character data is represented in memory using either the Extended ASCII or UNICODE table.*

1.9.2 Representing Translated Instructions

The technique used to represent translated instructions in memory is the same technique used to represent characters in memory. A table is composed containing all of the possible translated instructions, and a unique bit pattern is assigned to each of them. For example, the bit pattern for the translated instruction to subtract two integers could be 01000000, and the bit pattern to divide two integers could be 01000010.

Unlike the Extended ASCII and UNICODE tables that are used by all computer systems to store characters, these translated instruction tables vary from one CPU to another. Not only do the bit patterns vary, but the number of bits used to represent a translated instruction also varies. The tables are platform dependent, which is the reason Java came into being. To determine the translated memory representation of a divide instruction on a particular platform, we have to look up the bit pattern for the divide instruction in the instruction table of the CPU of that platform.

For the Java Virtual Machine, each translated instruction is assigned an eight-bit pattern. Because the patterns consist of eight bits, or one byte, the patterns are called byte codes. Table 1.1 gives the Java byte codes for the translated integer arithmetic instructions: add, subtract, multiply, and divide. As indicated in this table, when the Java Virtual Machine receives a byte code of 01101100, it performs a divide operation.

Table 1.1
Java Byte Codes for Integer Arithmetic Instructions

Instruction	Java Byte Code
add	01100000
subtract	01100100
multiply	01101000
divide	01101100

! TIP

Translated Java instructions are represented in memory using patterns of eight bits called byte codes.

1.9.3 Representing Numeric Data

Unlike the two previously described schemes, the scheme used to represent numeric data does not use a table because, for one thing, the table would be infinitely long. Rather, the scheme is based on the theory of numbers. All number systems have a base. Our number system's base is 10, which anthropologists speculate is due to the fact that we have ten fingers and ten toes. In number theory, the base of a number system determines the number of digits in the system. Because our number system is base 10, it has 10 digits (0 through 9). Conversely, the theory of numbers tells us that if a number system has 10 digits, its base is 10.

Armed with this knowledge of number systems, it was decided that numeric data would be represented in memory using a number system whose base is 2 because one bit can represent the system's two digits: 0 and 1*. Anthropologists would tell us that a base-2 number system would probably be our number system if we had two fingers. Because we do not have two fingers, we need to understand how to convert from base 2 to base 10 to interpret what base-10 number a bit pattern represents and how to convert numbers from base 10 to base 2 so we can store numbers in memory.

! TIP

Numeric data (data that will be used in a mathematical expression) is represented in memory using a binary number system.

Fundamental to these conversions is the realization that digit position values in a base-2 number system are not the same as in our base-10 system. Starting from the right, the digit position values in our number system are the 1s position, the 10s position, the 100s position, etc.

*John von Neumann, often called the father of the modern computer, originally proposed this scheme.

These represent 10^0, 10^1, 10^2, etc. Extrapolating this to a base-2 system, the digit position values starting from the right are 2^0, 2^1, 2^2, etc. Figure 1.33 gives the first eight digit position values of the base-2 number system with their decimal (base 10) equivalent below them. Knowing the digit position values of the binary number system, we can now convert from base 2 to base 10, and base 10 to base 2.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
128	64	32	16	8	4	2	1	base-2 position values

Figure 1.33
First eight digit position values of the binary number system.

To convert a base-2 representation (e.g., 01000010) of an integer numeric value stored in memory to base 10, we simply write the bit pattern below the base-10 digit position values that are shown in Figure 1.33 and add the values that have a 1 under them. For example, for the bit pattern 01000010, the process would be:

128	64	32	16	8	4	2	1	base-2 position values
0	1	0	0	0	0	1	0	internal representation
	64					2		

Therefore, 01000010 represents the base-10 number 66 (64 + 2).

This conversion process implies that the bit pattern 11111111 represents the largest integer that can be represented using 8 bits, which is the base-10 number 255 (255 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1). To represent integers larger than 255, more bytes of storage would be dedicated to each integer numeric value.

To convert a base-10 integer to its binary bit pattern to store the numeric value in memory, we begin by writing out the base-10 digit position values that are shown in Figure 1.33. Then starting on the left, we place a 1 under all of the position values that when added together give the base-10 number. The remaining position values are filled in with zeros.

To quickly determine which positions that should have a 1 placed under them, use the following algorithm until the right most position value is reached:

1. Let `n` (e.g., 66) be the base-10 integer value to be represented in memory
2. Start at the left most bit, `b`
3. Set `v` to `b`'s position value (e.g., v = 128)
4. If (`n` – `v`) is positive or equal to zero then:
   a. Place a 1 under `b`'s position
   b. Set `n` = (`n` – `v`)
   Else place a 0 under b's position
5. Move b to the next bit to the right
6. Go to step 3

Table 1.2 illustrates the use of this algorithm to convert 66 to its 8-bit binary representation. Each row in the table represents an execution of steps 3 and 4 of the algorithm.

Table 1.2
Conversion of the Integer 66 to its 8-Bit Binary Representation

n	b	v	n – v	Binary Representation of n
66	7	128	–62	0
66	6	64	2	01
2	5	32	–30	010
2	4	16	–14	0100
2	3	8	–6	01000
2	2	4	–2	010000
2	1	2	0	0100001
0	0	1	–1	01000010

Before we conclude our discussion on how numeric data is stored in memory, we should comment on how negative integers and numbers with fractional parts, which are called real numbers in mathematics, are represented in memory. The short answer is that negative integers are represented using a scheme named twos complement form, and numbers with fractional parts are represented in a standardized[6] form analogous to scientific notation (as when 235.2374 is expressed as 2.352374×10^2). The details of these schemes are beyond the scope of this text, however, an understanding of the representation of positive integers as binary numbers discussed in this section is fundamental to an understanding these two representation schemes.

Finally, consider a byte of storage that contained the bit pattern 01000010. When we attempt to determine what is stored in this byte, a dilemma arises. If we look into the Extended ASCII table we would conclude the character B is stored there. We have also learned that this could also be the base-10 integer 66. It is also the byte code instruction to store an integer in RAM memory. To resolve these kinds of dilemmas, the language translator keeps track of the types of information that is stored in various parts of RAM. If we knew that the bit pattern 01000010 was in the area of RAM where characters are stored, then it represents character B.

1.10 CHAPTER SUMMARY

In this chapter, you learned about the hardware and software components of a computer system, how they are arranged, and how they interact with the user. The hardware components consist of the central processing unit, memory, and input/output devices. Main or RAM memory interacts with the CPU and stores the data and instructions that are about to be processed by the CPU. The backing store or secondary memory, such as a hard drive, stores data and instructions more permanently.

The modern computer was developed over centuries through the efforts of many people. It has become smaller, faster, cheaper, and more reliable as it evolved from a room-sized device to the small hand-held mobile and wearable devices common today.

Java is an object-oriented programming language that allows a programmer to represent and process real-world objects within application programs and computer games. Classes are the templates for creating objects, which contain both data and methods to operate on the data. All information contained in a computer is represented in binary as translated instructions, numeric data, and character data. Java programs are translated into byte codes, which can be executed by the Java Virtual Machine, making them platform independent and portable.

New programs are defined in a written specification, then the program's algorithms are discovered and an IDE is used to compose and test the program. Game programs are more easily composed by importing a game environment into the program, such as the one contained on the DVD that accompanies this textbook. Game environments supplement the Java API by providing methods that perform tasks common to most games, such creating an interactive game board on which the game objects can be drawn and moved.

The discovery of a program's algorithms is usually the most difficult part of producing a new program. Throughout this text, we will use game programming to illustrate the use of programming concepts and use game algorithms to introduce the reader to the algorithm discovery process.

Knowledge Exercises

1. Between 1989 and 2004, the number of computers per 1,000 U.S. citizens increased by a factor of approximately:

a) 2 **b)** 4
c) 8 **d)** 12

2. What is the difference between hardware and software?

3. Explain the difference between operating systems and application programs.

4. Which of the following characteristics are associated with RAM (main) memory?

a) Nonvolatile **b)** Very fast
c) Very large capacity **d)** Expensive

5. Which of the following characteristics are associated with backing (secondary) storage?

a) Nonvolatile **b)** Very fast
c) Very large capacity **d)** Expensive

6. Give three examples of:

a) Input devices **b)** Output devices
c) Backing (secondary) storage devices

7. Some computer devices have a single use while others have multiple uses.

a) Name a device that is only used for output.
b) Name a device that is only used for input.
c) What device can be used for both input and output?

8. Name and explain the function of each of the three major hardware components of a computer system.

9. How would you respond to a friend who asked you who invented the computer?

10. Examples of operating system programs include all of the following except:
    a) MAC OS
    b) Windows
    c) Java
    d) Linux

11. Volatile memory refers to memory that:
    a) Permanently stores data
    b) Loses its contents if power is interrupted
    c) Is added to the computer externally

12. Word processing, e-mailing, and searching the Web are all examples of using:
    a) Application software
    b) Systems software
    c) Programming
    d) None of the above

13. Which of these replaced vacuum tubes in second-generation computers?
    a) Paper tape
    b) The mouse
    c) Chips
    d) Transistors

14. Who developed assembly language, the first compiler, and the language COBOL?
    a) Alan Turing
    b) Ada Lovelace
    c) Grace Hopper
    d) John von Neumann

15. Name the person referred to by each of these titles or descriptions:
    a) First programmer
    b) Inventor of the Java programming language

16. Give the four features of a program that are identified in its specification.

17. What is meant by platform independence?

18. True or False: To achieve platform independence, Java byte codes are translated on the end user's computer system.

19. What is the difference between a class and an object?

20. In a video game, a paddle will be used to reflect a ball into a pile of 200 bricks.
    a) How many objects will be involved in the game? What are they?
    b) How many classes will be defined in the program? Name them.

21. Give the terms that are represented by the following acronyms:
    a) CPU
    b) RAM
    c) I/O
    d) IDE
    e) JVM
    f) API
    g) GUI

22. Which of these refers to the process of breaking a problem into smaller parts in order to solve or program it?
    a) Divide and conquer
    b) Platform independence
    c) Portability
    d) Translation

23. The upper left corner of the game environment's game board is located at the (x, y) pixel coordinates:

a) (0, 0) **b)** (500,500)
c) (622,535) **d)** (5, 30)

24. Which of these is not a component of a typical game program?

a) Score **b)** Time limits
c) Napier's bones **d)** Game piece objects

25. Name the three types of information represented in memory.

26. Write the 8-bit binary equivalent for each of these base-10 numeric values:

a) 51 **b)** 77
c) 115 **d)** 131
e) 227 **f)** 254

27. Write the base-10 (decimal) equivalent number for each of these binary values:

a) 01010011 **b)** 00101111
c) 00000000

28. Give the 8-bit memory representation of the characters C and c.

Preprogramming Exercises

1. Think of a video game and conduct a conversation with yourself that includes the features common to most games that are tabulated in Figure 1.1.23. Based on that conversation, write a specification for the game that gives the game's name, the task or objective of the game, and a description of the inputs and outputs. The game must include at least two different types of game objects and one of the objects has to be controlled by the user via the cursor control keys and the game board directional buttons.
2. Logan is a teacher with 25 students in his class. Write a specification for a program that will show Logan the lowest, highest, and average class grades on an examination.
3. Using the template given in Figure 1.31 and the directions given in Section 1.8.5, write the line of code necessary to change the game window's size to 800x600 with the new title "My Great Game Window."

Enrichment

In the same way that computers and programming languages have evolved over time, game programs also have developed from very simple games to the present multiuser, interactive games. Search the Internet to discover some of the historical developments of computer games. Some of the questions you might research are:

- When and where the first games were developed
- What companies were created for developing games

- Who are the leaders today in the field of games
- How do today's games differ from the earliest computer games

(Be sure to record the sources of your information.)

References

Fullerton, Tracy, *Game Design Workshop*, 2nd ed. Burlington, MA: Morgan Kaufman Publishers, 2008.

Iverson, Jakob, and Michael Eierman. *Learning Mobile App Development*. Upper Saddle River, NJ: Addison-Wesley, 2013.

Lucci, Stephen, and Danny Kopec. *Artificial Intelligence in the 21st Century.* Dulles, VA: Mercury Learning and Information, 2013.

Swade, Doron. *Charles Babbage and his Calculating Machines*. London: Science Museum, 1998.

Video

"Queen of Code", the story of Grace Hopper, *https://vimeo.com/118556349.*

Endnotes

1 http://en.wikipedia.org/wiki/History_of_computing

2 http://www.computersciencelab.com/ComputerHistory/History.htm

3 http://www.webopedia.com/DidYouKnow/Hardware_Software/2002/FiveGenerations.asp

4 http://community.seattletimes.nwsource.com/archive/?date=19961124&slug=2361376

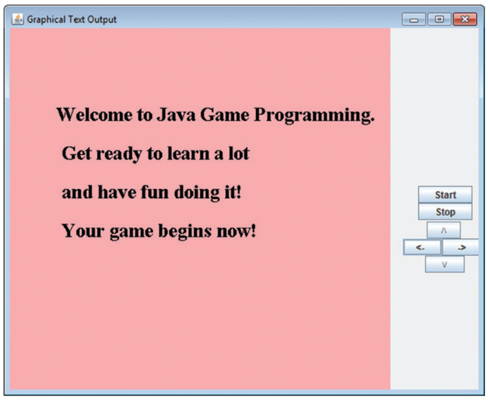
5 Decker, Rick and Stuart Hirschfield. *The Analytical Engine. Belmont, CA. Wadsworth Publishing Company, 1992, p.17 (Now online: http://www.course.com/downloads/computerscience/aeonline/)*

6 The standard is named IEEE 754-2008

7 *http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-7.html*

Chapter 2

Variables, Input/Output, and Calculations



In this chapter

In this chapter, you will learn how to use the basic Java program template to develop a program that performs input, mathematical calculations, and output. Various methods to facilitate meaningful input and understandable output will be introduced, as will techniques for storing data in RAM memory and performing mathematical calculations that go beyond basic arithmetic operations. All these techniques are used in most programming applications.

After successfully completing this chapter you should:

- Understand the basic components of a Java program
- Recognize the difference between primitive and reference variables and how they store data
- Be able to declare and use variables in a program
- Perform input from a dialog box
- Perform output to a dialog box, as well as output to the system console and a graphical window
- Use arithmetic calculations and mathematical functions and constants in the Java Math class
- Perform basic formatting of numeric output
- Understand and be able to use the counting algorithm
- Apply these concepts to begin producing a computer game

2.1 THE JAVA APPLICATION PROGRAM TEMPLATE

If you were writing a letter to you friend Sally, it would probably begin with an opening salutation, for example, "Dear Sally," and end with a closing salutation, for example, "Sincerely," followed by your signature. Opening and closing salutations are usually considered to be a minimum template for any letter we compose. In between these salutations, we would put the text specific to the letter we are writing.

Similarly, most programming languages have templates for composing a program in that language. These templates begin and end with text specific to the language, and we write, or *code*, the instructions specific to the program we are composing inside the template. The minimum template for a Java program is depicted in the top half of Figure 2.1.

```
1 public class ProgramName
2 {
3    public static void main(String[] args)
4    {
5
6    }
7 }
```

Java Program Template

```
1 public class ProgramName
2 {
3    public static void main(String[] args)
4    {
5       System.out.println("Hello");
6    }
7 }
```

Java Program to Output the Word "Hello"

Figure 2.1
Template of a Java program and a program that outputs the word "Hello."

The phrase `ProgramName` on line 1 of this template is replaced with the name of the program being composed, and the instructions specific to the program are placed within the program's *code block,* within the braces that appear on lines 4 and 6. For example, in the bottom half of Figure 2.1 an instruction, or *executable* statement, has been added to line 5 of the template to produce a program that outputs the word "Hello."

NOTE *All Java* **executable** *statements end with a semicolon.*

When a Java program is run, the first instruction to execute is always the first executable instruction coded after the open brace on line 4. In programming jargon this statement is said to be the program entry point, and all programming languages designate a location in the program template to be the program's entry (starting) point. By default, the program statements that follow the program entry point usually execute sequentially in the order they appear in the program.

If an Integrated Development Environment (IDE) is being used to compose a program, it will normally ask for the name of the program (or project). Then the IDE generates the code template with the phrase `ProgramName` on line 1 replaced with the program's name. In addition to the seven lines shown in the top of Figure 2.1, some IDEs add several other lines to the template, the most common of which is a statement on line 5 to output the phrase "Hello World." However, for the template to be grammatically correct, all IDEs will include the seven lines shown in the top portion of Figure 2.1.

2.2 VARIABLES

Most programs process data that is input to the program. For example, a program may compute the sum of two input bank deposits. To be processed, data must be stored in the memory of the computer system. All programming languages contain statements for defining *variables,* which are memory cells that can store one piece of data. Before a variable can be used in a Java program, it must first be declared. When a variable is defined, or declared in a program, the programmer assigns it a name and designates the type of information to be stored in the cell. For example, a variable named `deposit` could be used to store the amount of a deposit, in which case its type would be a number with a fractional part.

Definition

A **variable** is a named memory cell that can store a specific type of data.

Variables must be declared before they can be used.

In Java, valid variable names must begin with a letter and cannot contain spaces. After the first letter, the remaining characters can be letters, digits, or an underscore. They cannot be Java key words. (See Appendix D.) Variable names that do not follow these rules are invalid and are identified by the Java translator as syntax (grammatical) errors. Good coding style dictates that variables begin with a lowercase letter, and new words in the variable name begin with an uppercase letter. In addition, the name of the variable should be representative of the data item being stored in the memory cell yet be as brief as possible. For example, a variable used to store the balance of my savings account could be named `myBalance`.

Good choices for variable names make our programs more readable. The variables on the left side of Figure 2.2 are well composed: they are syntactically correct, use good naming conventions, and imply what they store. The variable names on the right side of the figure are not well composed, concise, or meaningful.

Well Composed	**Poorly Composed**
`firstName`	`Fst`
`deposit1`	`theFirstOftheBankDeposits`
`myBalance`	`mb`
Valid	**Invalid**
`zipCode`	`zip  Code`
`phoneNum`	`phone#`
`grade1`	`1stGrade`

Figure 2.2
Variable names.

The information stored in a memory cell can change or vary during the execution of the program (which is why these storage cells are call variables). However, once designated, the *type* of the information stored in the memory cell (e.g., a number with a fractional part) cannot be changed.

In Java, there are two kinds of variables: **primitive** variables and **reference** variables. The type of data stored in primitive variables can be a single numeric data value, one character, or one Boolean truth value. Reference variables store RAM memory addresses. The grammar, or syntax, used to declare a primitive variable is the same grammar used to declare a reference variable. In the next section, we will discuss this syntax and the use of primitive variables in our programs. The use of reference variables will be discussed in Section 2.5.

Definition

Primitive variables store one numeric value, one character, or one truth value.

Reference variables store memory addresses.

2.3 PRIMITIVE VARIABLES

The Java statement used to declare a variable begins with the type of the information stored in the variable, said to be the variable's *type*, followed by the name of the variable. Like all Java statements, variable declaration statements end with a semicolon. Optionally, the declaration statement can also include the value to be initially stored in the variable. If the initial value is not specified within the variable declaration statement, the variable is set to a default value. Default values are dependent on the type of information stored in the variable. For example, the statements

```
double deposit;
double price = 5.21;
```

declare the variables named `deposit` and `price`, with `deposit` initialized to the default value 0.0 and `price` initialized to the value 5.21. The word **`double`** is a *keyword* in Java.

In programming languages, keywords are words that have special meaning to the translator that translates program statements into the language of the computer system. The keyword **`double`** means that the memory cell being defined will store a number with a fractional part and the size of the storage cell will be 8 bytes (64 bits). Table 2.1 gives the keywords used to specify the type of a primitive variable. The size of the storage cell implied by the use of the keywords is also given. As noted in the table, the keywords used to declare integer and real numeric types are different, and the size of the storage cell limits the numeric range and precision of the stored numeric value.

When storage is not at a premium, it is best to use the type **`int`** for integer variables because most programs deal with integers within the range -2,147,483,648 to +2,147,483,647. If a data item were beyond that range, it would not be properly represented in an **`int`** variable. For larger integer values, the type **`long`** should be used. Integer data beyond the range of the type **`long`** cannot be stored in a primitive variable. The Java API class `BigInteger` provides a remedy for this situation and will be discussed in Chapter 7, "Methods and Objects: A Second Look."

Table 2.1
Primitive Data Types

Data Type	Key Word	Cell Size (bytes)	Range and Precision
Integer Numeric	`byte`	1	-128 to +127
	`short`	2	-32,768 to +32,767
	`int`	4	-2,147,483,648 to +2,147,483,647
	`long`	8	-9,223, 372,036,854,775,808 to +9,223, 372,036,854,775,807
Real Numeric	`float`	4	±1.40129846432481707 E-45 to ±3.4028234663852886 E+38 *(7 digits of precision)*
	`double`	8	±4.94065645841246544 E-324 to ±1.797693134862157 E+308 *(15 digits of precision)*
Truth Value	`boolean`	1	`true` or `false`
One Character	`char`	2	Upper and lowercase keyboard characters and other entities (see Appendix C)

When storage is not at a premium, it is best to use the type `double` for variables that will store real numbers (numbers with fractional or decimal parts) because the range of the real numbers processed by a program is usually within the range of the type `double`. As is the case for large integers, Java provides an API class (`BigDouble`) for storing real numbers whose range exceeds that of a double. In addition, because numeric literals, (e.g., 1.5) are represented as type double, an `f` (for float) must be added to the end of an initial value in a float variable declaration to inform the translator that the loss of precision is acceptable:

```
float change = 1.5f;
```

When the initial value of a character variable is specified, it is enclosed in single quotation marks, and the initial values of Boolean variables begin with a lowercase letter:

```
char myFirstIntial = 'W';
boolean isRaining = false;
```

Multiple variables of the same type can be declared in a single Java statement. The variables are separated by commas, and the statement cannot include initial values:

```
short n1, n2, n3;
boolean isRaining, isSnowing;
char letter1, letter2, digit1, digit2;
```

When initial values are not specified in a variable declaration statement, the variables are set to default values. The default value for the integer types (`byte`, `short`, `int`, and `long`) is zero, and the default value for the real types (**`float`** and **`double`**) is 0.0. For the Boolean type (**`boolean`**), the default value is **`false`**, and for the character type (**`char`**), it is `''`. The values **`true`** and **`false`** are Java keywords.

2.4 SYSTEM CONSOLE OUTPUT

The system console is a window that a program can use to communicate with the user of the program. When information flows from the program to the system-console window, we say that the program is performing output. Conversely, when the information flow is from the system-console window to the program, we say the program is performing input. For brevity, these information transfers are referred to as console input and console output, respectively, or more simply console I/O. In this section, we will discuss console output, and console input will be discussed in Chapter 4 "Boolean Expressions, Making Decisions, and Disk I/O."

The two Java statements used to perform output to the system console are:

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

Like all Java executable statements, they both end with a semicolon. The output item, which is referred to as an argument, is coded inside the statement's open and close parentheses. The only difference between these two statements is that the first one leaves the console's cursor at the end of the output item, and the second one positions it at the beginning of the next line.

2.4.1 String Output

Technically speaking, the item to be output must be a sequence of characters, which in programming languages is called a *string* (e.g., This is Console Output). In Java, strings can either be *string literals* or `String` objects. String objects will be discussed in Section 2.5.

String literals are strings enclosed in double quotes. To output "*This is Console Output*" we would code the string literal `"This is Console Output"` inside the parentheses of a console output statement. The following code fragment would display two lines of output:

```
System.out.println("This is Console Output");
System.out.print  ("from the program");
```

The first line would contain *This is Console Output*, and the second line would contain *from the program*. Because the second line is a `print` statement, the console's cursor would appear on the second line just after the word program.

2.4.2 The Concatenation Operator and Annotated Numeric Output

The concatenation operator, which is coded as a plus (+) sign, can be used to combine two strings into one. The statement

```
System.out.println("Hello " + "World");
```

produces the output *Hello World* to the system console. Before the output is performed, the first string literal, containing the word "`Hello `", is combined with the second string literal "`World`". The resulting string, "`Hello  World`", is then output to the console. There is no limit to the number of string literals that can be combined using concatenation operators to produce the string argument output by the `print` and `println` methods. The statement

```
System.out.println("Hello " + "World," + " I'm Bill.");
```

produces the console output *Hello World*, *I'm Bill.*

To make the output of numeric data more meaningful and user-friendly, the output should always be identified or annotated. For example, the output *The price is $5.21* is much more informative than the output *5.21* The annotation *The price is $* can be included in the output using the concatenation operator.

```
System.out.println("The price is $" + price);
```

The Java translator interprets the plus sign used in this context as the concatenation operator because the item to its left is a string literal. It will fetch the contents of the variable `price`, convert it to a string, and then concatenate that string with the previous string literal. The resulting console output is "The price is $5.21."

Because there is no limit to the number of string literals that can be combined to produce the string argument of the `println` and `print` methods, the annotated contents of several variables can be output to the console using one console output statement. The console output *The price of the 10 items is $5.21.* is produced by the code fragment:

```
int quantity = 10;
double price = 5.21;
System.out.println( "The price of the " + quantity +
                    " items is $" + price + ".");
```

The indentation in the above `System.out.println` statement has been used to improve its readability. It prevents the statement from going beyond the eightieth column and is considered good programming practice.

2.4.3 Escape Sequences

It is often necessary to output strings containing characters that have special meaning to the Java translator. For example, a double quotation mark (") is meant either to begin or end a string literal, and a single quote (') is meant to begin or end a character literal. Suppose we wanted to output *Joe said, "Hello"* followed by a period. To be grammatically correct in English, the word Hello has quotes around it because it is something Joe said. However, the output statement

```
System.out.println("Joe said, "Hello".");
```

would result in a syntax error because a double quotation mark in Java is meant to be either the beginning or end of a string literal. Therefore, the translator would assume the quotation mark preceding the word `Hello` was meant to terminate the string literal, which began with the quotation mark preceding the word `Joe`. Under this assumption, the translator expects the next character to be a close parenthesis followed by a semicolon, or a concatenation operator. Instead, it finds the character `H`, which produces a syntax error.

To solve this problem, Java provides escape sequences, which are a sequence of two characters coded inside a string literal. The first character in the sequence is always the backslash (\) character. When the translator encounters a backslash inside a string literal, it always considers this to be the beginning of an escape sequence and effectively looks up the meaning of the escape sequence, given in Table 2.2. In other words, the backslash tells the translator to **escape** from its normal way of interpreting this backslash and the next character, and instead look into the table of escape sequences for the meaning of these two characters.

For example the escape sequence `\"` (coded inside a string literal) means don't interpret the quotation mark as the beginning or end of a string literal but output a quotation mark. Therefore, the syntactically correct way to output the sentence *Joe said, "Hello".* is:

```
System.out.println("Joe said, \"Hello\".");
```

Now the quotation mark preceding the `H` in `Hello` is part of the escape sequence to output a quotation mark. It is not interpreted as the close of the string literal, which began with the quotation mark preceding the word `Joe`. Another escape sequence is coded after the `o` in `Hello` for the same reason. Proceeding to the right in the string literal, the translator encounters the quotation mark that follows the period, which it correctly interprets as the close of the string literal.

Because a backslash inside a string literal is interpreted as the beginning of an escape sequence, one obvious question is "how would we output a backslash?" The answer is that there is an escape sequence for outputting a backslash, which is a double backslash. The statement

```
System.out.println("Down \\/    Up /\\");
produces the output:  Down \/    Up /\.
```

The escape sequence `\'` is used to output a single quotation mark, and the escape sequence `\n` causes the cursor to move to a new line before completing the output. The escape sequence `\t` tabs the cursor to the right; this is useful when you want output to appear in columns. A list of the escape sequences is shown in Table 2.2.

Table 2.2
Escape Sequences

Escape Sequence	Sequence Name	Meaning
`\"`	Double quote	Output the double quotation mark (") character
`\\`	Backslash	Output the backslash (\) character
`\'`	Single quote	Output the single quotation mark (') character
`\b`	Backspace	Move the cursor back one character position
`\t`	Horizontal tab	Move the cursor to the next horizontal tab position
`\n`	New line	Move the cursor to beginning of the next line
`\r`	Carriage return	Move the cursor to beginning of the current line
`\f`	New page (form feed)	Move the cursor to the top left of the next page

The application shown in Figure 2.3 illustrates the declaration and initialization of primitive variables and the use of string literals and escape sequences to output the data stored in these variables. The program's outputs are included in the figure after the program's code.

```
1   public class ConsoleOutput
2   {
3     public static void main(String[] args)
4     {
5       // Primitive variable declarations
6       int age = 21;
7       double weight = 185.25;
8       boolean isRaining = false;
9       char letter1 = 'A';
10
11      System.out.println("The Program's Output Appears Below");
12      System.out.println("\t\t\"Hello World!\"");
13      System.out.print("\nJohn is " + age + " years old");
14      System.out.println(" and weighs "  + weight + " pounds");
15      System.out.println("Today it is " + isRaining +
16                         " that it is raining\n");
17      System.out.println("The first letter of the alphabet is " +
18                          letter1);
19
20      System.out.println("1/2 + 1/4 = 3/4");//blank lines are ignored
21    }
22  }
```

Program Output

The Program's Output Appears Below
"Hello World!"

John is 21 years old and weighs 185.25 pounds
Today it is false that it is raining

The first letter of the alphabet is A
1/2 + 1/4 = 3/4

Figure 2.3
The application **`ConsoleOutput`** and the output it produces.

! TIP

It is good coding style to declare all variables at the beginning of a program.

Lines 6-9 declare and initialize four different types of primitive variables. Lines 11 and 12 produce the first two lines of the program's output. Each statement contains one string literal. The string literal on line 12 begins with two tab escape sequences, which are used to center the second line of output under the first. In addition, `Hello World!` coded on line 12, is surrounded by two double-quote escape sequences, which produces the quotation marks on the second output line.

Lines 13–18 output the variables declared and initialized on lines 6-9. A new-line escape sequence begins the first string literal on line 13, which produces the blank line that precedes the third line of text output. Two concatenation operators are used on line 13 to combine the two string literals and the contents of the variable `age` after it is converted to a string. Line 14 uses similar operations to annotate the output of John's age and weight. The output displayed by lines 13 and 14 appear on the same line because line 13 uses a `print` rather than a `println` statement. As a result, the cursor is not advanced to the beginning of the next line after line 13 completes execution, which causes the output produced by line 14 to begin immediately after the word old. One subtlety on line 14 of the program is that its string literal begins with a space. This space becomes the space that separates the word *old* from the word and in the output produced by lines 13 and 14.

Lines 15-16 produce the next line of output, which contains the string version of the contents of the Boolean variable `isRaining`. They also produce the next blank line of output because the last string literal ends in a new-line escape sequence. The final two lines of output are produced by lines 17-18, which output the contents of the character variable `letter1`, and line 20, which outputs a single string literal.

Comments and Blank Lines

Line 5 of the program contains a single-line comment. A single-line comment begins with two forward slashes (//) and is terminated by a new line (`Enter`) keystroke. Comments are added to a program to improve the program's readability; they are ignored by the translator.

! TIP

It is good practice to include comments in your program to describe the portions that are not obvious to the reader.

A second comment appears at the end of line 20, stating blank lines (e.g., lines 10 and 19) in a program are ignored by the translator. It is good programming practice to separate major portions of a program with a blank line. This technique, like comments, improves the readability of a program. We will see more examples of the use of blank lines in a program later in this chapter.

2.5 STRING OBJECTS AND REFERENCE VARIABLES

As previously mentioned, in Java there are two kinds of variables: primitive variables and reference variables. Primitive variables store numeric, character, or Boolean data values. Reference variables store memory addresses. These addresses are the addresses of memory resident programming constructs called objects, and the contents of a reference variable is used to locate a particular object. We say they refer to an object, which is how they get their name, reference variables.

Suppose that we were writing a program and we wanted store the string `John` in memory. Based on what we have learned about primitive variables, we should declare a string variable, perhaps named `firstName`, and then initialize it to the string `"John"`. Unfortunately, Java does not contain string type variables, so the statement

```
string firstName = "John";  // error
```

is grammatically incorrect. However, there are `String` objects in Java. A `String` object can store a sequence of characters, and the address of the object can be stored in a reference variable. We begin by declaring a `String` reference variable that will store the address of our String object. Then we store the address of a newly created `String` object, containing the string `"John"` in the reference variable:

```
String firstName;
firstName = new String("John");
```

As we will learn in Chapter 3 "Methods, Classes and Objects: A First Look," this two-line grammar can be used to create objects in any class. For example, `Starship` objects, `Snowman` objects, or `Paddle` objects can be created simply by replacing the word `String` on both lines with the class names `Starship`, `Snowman`, or `Paddle`, and replacing the string "John" with something more relevant to these objects. The first line creates an uninitialized reference variable that, like uninitialized primitive variables, is set to a default value. The default value for reference variables is **`null`**. When the variable is a `String` reference variable, we say that the reference variable stores the **`null`** string.

! TIP "String s contains the null string," means that s stores a **`null`** value.

Because strings are so commonly used in programs, Java provides a simplified one-line grammar for creating and initializing `String` objects:

```
String firstName = "John";
```

NOTE *The abbreviated grammar to declare a string* `String` *object is:*
String referenceVariableName = intialStringLiteralValue;

This one-line grammar can only be used to create `String` objects and is modeled on the grammar used to declare and initialize primitive variables. Although the grammar is very similar, we must keep in mind that unlike the primitive variable `age`, initialized to store the value `21` on line 6 of Figure 2.3, the reference variable `firstName` is not initialized to the string `"John"`. Rather, the reference variable `firstName` stores the address of and refers to the `String` object that is initialized to the string `"John"`. Figure 2.4 shows the statements used to allocate memory to primitive variables and objects/reference variables. The arrow in the figure indicates that the reference variable `firstName` refers, or points, to the object.

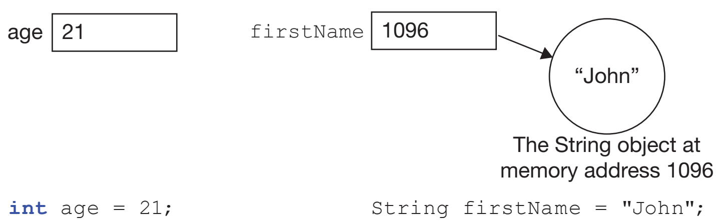


Figure 2.4
Memory allocated to primitive variables and objects/reference variables.

In addition to providing a simplified grammar for creating `String` objects, Java also provides a simplified grammar for outputting the strings contained inside these objects. Once again, it is modeled after the grammar used to output primitive variables. To output the string contained in a `String` object, we simply code the name of the variable that refers to the object. For example, the following code fragment produces the output *My name is John Smith, my age is 21* on the system console:

```
int age = 21;
String firstName = "John";
String lastName = "Smith";
System.out.print ("My name is "  + firstName  + " " + lastName);
System.out.println (", my age is "  +  age);
```

The differences between the way Java stores primitive data items and string data items can be ignored when writing variable declaration statements and output statements. As we will see in Chapter 3, these differences cannot be ignored for any other kind of object.

2.6 CALCULATIONS AND THE MATH CLASS

The first operational computers were used by mathematicians to compute the values of equations, which is how they obtained their name *computers*, and a significant portion of the processing that modern computers perform is still calculations. Java, like most programming languages, provides the ability to perform basic arithmetic calculations and provides additional features to perform more complex calculations. This section begins with a discussion of how to incorporate basic arithmetic calculations into a Java program and then discusses how to incorporate commonly used mathematical constants and functions into these calculations.

2.6.1 Arithmetic Calculations and the Rules of Precedence

Arithmetic calculations are performed in Java using arithmetic expressions. Arithmetic expressions consist of a series of operands separated by operators. In the simplest case, the operands are numeric constants, and the operators are the four arithmetic operators: add subtract, multiply, and divide. For example, 10 + 21 – 5 is a simple arithmetic expression that evaluates to 26. Generally, simple arithmetic expressions are evaluated from left to right. The addition would therefore be performed before the subtraction.

The symbols used for the four arithmetic operators are given in Table 2.3. The third entry in the table, the modulo (or mod) operator, is used to find the remainder in division. For example, 14 % 3 evaluates to 2. All of the operators can be used with integer or real operands.

In addition to numeric constants, called numeric literals, operands can be the names of variables that store numeric values. When a memory cell name is used in an arithmetic expression, the value stored in the memory cell is fetched, substituted for the memory cell name, and the arithmetic expression is evaluated. For example, given the variable declarations:

```
int x = 10;
int y = 29;
int z = 5;
```

Table 2.3
The Java Symbols for the Arithmetic Operators (In Order of Precedence)

Arithmetic Operation	Java Symbol
multiply	*
divide	/
Modulo or mod	%
add	+
subtract	-

the arithmetic expression x + y – z evaluates to 34. A mix of numeric literals and memory cell names can be used as the operands in any arithmetic expression, so the expression x + 29 – z is a valid arithmetic expression that also evaluates to 34.

An arithmetic operation performed on two integers always results in an integer value, and an arithmetic operation performed on two real values always results in a real value. When one operand is an integer and the other is a real value, the result is always a real value, and the arithmetic is referred to as *mixed mode* arithmetic. Before mixed mode arithmetic is performed, the integer value is converted to a real value (e.g., 10 becomes 10.0).

Integer Division

When the two operands are integers and division is performed, the results are sometimes surprising. That is because the division of two integers always produces an integer result that is truncated and not rounded. For example, given the variable declarations

```
int x = 10;
int y = 29;
int z = 5;
```

the following arithmetic expressions would evaluate to the values on the far right side of each expression:

x / z = 10 / 5 = 2
y / x = 29 / 10 = 2 (0.9 lost, due to truncation)
z / x = 5 / 10 = 0 (the most surprising result, 0.5 is truncated to zero)

Precedence Rules

Consider the arithmetic expression 10 + 6 – 2. The expression evaluates to 14 whether we perform the addition first (16-2) or the subtraction first (10+4). Similarly, the expression 10 * 6/2 evaluates to 30 whether we perform the multiplication first (60/2) or the division first (10*3). In both cases, the value of the expressions is independent of the order in which we apply the arithmetic operators. In general, if an expression contains just addition and subtraction operators, or contains just multiplication and division operators, then the evaluation of the expression is independent of the order in which we apply the arithmetic operators. Java considers these expressions to be simple arithmetic expressions and, as previously stated, they are evaluated from left to right.

This is not the case for arithmetic expressions that mix addition and/or subtraction operators with multiplication and/or division operators. Consider the expression 10 + 6 * 2, which performs addition and multiplication. If we perform the addition first, the expression evaluates to 32 (16 * 2), but if the multiplication is done first it evaluates to 22 (10 + 12). The arithmetic expression appears to be ambiguous. Fortunately, mathematicians have stipulated a way of resolving the ambiguity called the *rules of precedence.* These rules state that multiplication and division are performed before, or take precedence over, addition and subtraction. Using this rule, the expression 10 + 6 * 2 evaluates to 22.

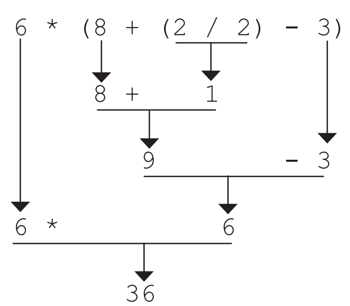
Operators that are performed first, such as multiplication and division, are said to have higher precedence. Table 2.3 lists the arithmetic operators in high-to-low precedence order, with multiplication and division being the highest precedence operators in the table, and addition and subtraction the lowest. The expression 5 * 7 % 2 would evaluate to 1 because multiplication is of higher precedence than the mod operator. Java contains other operators, for example logic operators, and each Java operator has been assigned a precedence level. A complete list of Java operators and their assigned precedence level is given in Appendix E.

NOTE *Java evaluates arithmetic expressions using this mathematical rule of precedence: multiplication and division are performed before modulo (mod) operations, which are performed before addition and subtraction operations.*

If we wanted the addition or subtraction in an arithmetic expression to be performed before multiplication or division, we would use a set of parentheses to override the precedence rules. The expression (10 + 6) * 2 would evaluate to 32. To average the numbers 2, 4, and 6, we would write (2 + 4 + 6) / 3, which would evaluate to the correct average 4 = 12 / 3. (Without the parentheses, only the 6 would be divided by the 3 because the division operation would be performed first.)

NOTE *Parentheses override the rules of precedence.*

In summary, the parts of an arithmetic expression inside parentheses are evaluated first using the rules of precedence to determine the order of the operations. If the operators are of equal precedence, they are evaluated from left to right. The following example, which contains a set of nested parentheses and evaluates to 36, illustrates this process.



2.6.2 The Assignment Operator and Assignment Statements

In Section 2.3, we learned that a variable named `price` could be declared and initialized to the value 5.21 by coding:

```
double price = 5.21;
```

The equals (=) symbol used in this declaration is called the *assignment operator* because it assigns values to memory cells. In this case, the memory cell named `price` is assigned the value 5.21. Although the statement should be read as "double price *is assigned* 5.21," most programmers would read it as "double price *equals* 5.21," which is unfortunate because (as we will see) the operator does not represent the mathematical concept of equality. Rather, it represents the flow of the data value on its right side (in the above statement, `5.21`) into the memory cell named on its left side, (in the above statement, `price`).

Assignment Statements

In addition to being used to initialize variables in a declaration statement, the assignment operator is also used in statements that reassign (actually overwrite) the contents of memory cells previously declared in a program. These statements are called assignment statements. For example, after the following two statements execute, the variable `price` stores the value `6.25`.

```
double price = 5.21;
price = 6.25;
```

In addition to being a numeric literal, the entity on the right side of the assignment operator can be an arithmetic expression. For example:

```
answer =  x + 21 - z;
```

When this is the case, we should realize that the execution of the statement is performed in three steps:

1. *Fetch* the contents of the variables coded on the right side of the assignment operator from memory and substitute these values into the arithmetic expression
2. *Evaluate* the arithmetic expression considering parentheses and the rules of precedence
3. *Store* the value of the arithmetic expression in the memory cell coded on the left side of the assignment operator

In an assignment statement, the item on the left side of the assignment operator must be the name of a variable. The statement cannot be reversed by placing the name of the variable on the right side of the assignment operator. That is,

```
answer =  x + 21 - z;
```

is *not* the same as

```
x + 21 - z = answer;
```

The second expression will produce a syntax error. Armed with this understanding, the assignment statement (which is probably executed on a person's birthday)

```
age = age + 1;
```

will increase the value stored in the memory cell `age` by one. In addition, a mathematician would now understand that the assignment operator does not represent equality and would not run from this statement in horror proclaiming that nothing (in this case age) could be equal to itself plus one. Surveys of programs conducted in the 1970s indicate that 47% of the statements contained in programs are assignment statements, so this is an important concept to understand.

2.6.3 Promotion and Casting

Generally speaking, the type of the value being assigned to a variable should match the type of the variable. When this is not the case, the Java translator checks this to make sure that there is no chance that part of the value being assigned to the variable could be lost when the value is stored in the variable. For example, if a real number (e.g., 2.7) was assigned to an integer-type variable, the fractional part of the value (0.7) would be lost. As a result, the statement below produces the translation error "possible loss of precision," because it assigns a **double** value (2.7) to an integer memory cell.

```
int newValue = 2.7;
```

! TIP Avoid assigning a numeric with a fractional part (e.g., types **float** and **double**) to an integer type variable because it will generate a translation error.

This statement does not produce a syntax error because performing an arithmetic operation on two integers (21 / 10) always results in an integer (in this case, 21 / 10 = 2).

```
int newValue = 21/10;
```

The Java translator also checks assignment statements to determine if the value being assigned to the variable, coded on the left side of the assignment operator, is within the variable's range. As shown in the right column of Table 2.1, the range of the numeric values that can be stored in a numeric variable depends on its type. Within the four integer types, the type `long` has the largest numeric range, and within the real types, the type `double` has the largest range.

The progression shown in Figure 2.5 summarizes the valid assignments between types (those that will not result in a loss of precision and guarantees that the range of the variable being assigned is large enough to store the value assigned to it). A valid assignment is when the type of the variable being assigned is to the left of the type of the value being assigned to it (e.g., a **double** variable can be assigned **int** values). When this is the case, we say that the value has been promoted to the type of the variable.

double ← **float** ← **long** ← **int** ← **short** ← **byte** ← **char**

Note: a valid promotion is from right to left (←)

Figure 2.5
Valid promotions of numeric types.

Although the types of the variables used in the code fragment in the assignment statements below are not the same, they are valid because they follow the promotion order given in Figure 2.5.

```
byte aByte = 20;
char a = 'a';
```

```
int anInt;
double aDouble;
anInt = aByte;
anInt = a;
aDouble = aByte;
aDouble = anInt;
```

Mixed Mode Arithmetic Expressions

Mixed mode arithmetic expressions are expressions in which the operands are not of the same type. To evaluate the terms of these expressions, the operand whose type is further to the right in Figure 2.5 is promoted to the type of the other operand, the term is evaluated, and the resulting value is in the promoted type. For example, the following code fragment contains a mixed mode arithmetic expression:

```
double salary = 523.56;
float raise = 1.1f;
salary = 10 + salary * raise;
```

The arithmetic expression in the assignment statement contains an integer literal, a **`double`** variable, and a **`float`** variable. During the evaluation of this expression, the value stored in `raise` would be converted to a double, and then the multiplication operation would be performed. The result would be a double value. That value would then be added to the integer literal 10 after it was converted to double. The resulting double value would be assigned to the variable `salary`.

Casting

One use of the word casting is the process of turning an entity into something it is not. For example, a frail mild-mannered actor could be *cast* into the role of a professional wrestler. In computer science, the term is used to describe the process of changing the type of a value to another type.

Changing the type of a variable or numeric literal in mixed mode arithmetic expressions previously discussed is an example of automatic casting. Even when an arithmetic expression is not a mixed-mode expression, there are times when it is desirable to cast operands into other types before the expression is evaluated. For example, a value that is an integer variable could be cast into a real value before it is used in an arithmetic expression. This is a very common use of casting.

Consider the calculation of the ratio of two integer variables `n1` and `n2`.

```
int n1 = 111;
int n2 = 10;
double ratio = n1 / n2;
```

The arithmetic expression will evaluate to an integer because both operands are integers. As a result, we will lose the factional part of the ratio, and the variable ratio will be assigned 11.0.

A situation that is more confusing is illustrated in the code fragment below. The integer denominator (`100`) is larger than the integer numerator (`90`). In this case, the variable `ratio` is always assigned 0.0.

```
int numberOfStudents = 100;
```

```
int numberPassing = 90;
double ratio = numberPassing / numberOfStudents;
```

To retain the factional part of a value calculated by dividing to integers, we change, or cast, the type of one of the operands into one of the real types (**double** or **float**). The syntax of this nonautomatic casting is to enclose the numeric type into which the operand is being cast inside of parentheses. The following fragment uses casting to change the fetched contents of the variable `n1` into a double before the arithmetic operation is performed. The outer set of parentheses in the third statement is necessary because arithmetic operators take precedence over casting.

```
int n1 = 111;
int n2 = 10;
double ratio = ((double) n1) / n2;
```

After casting is performed, the arithmetic expression involves a double value (111.0) and an integer variable (`n2`): a mixed mode expression. Automatic casting then converts `n2` to a double, and then the division is performed that produces a double (11.1). The value 11.1 is assigned to `ratio`. In this code fragment, `ratio` would be assigned the value 0.9.

```
int numberOfStudents = 100;
int numberPassing = 90;
double ratio = ((double) numberPassing) / numberOfStudents;
```

NOTE *Arithmetic operators take precedence over nonautomatic casting.*

Another common use of casting is to inform the translator that you want to violate the promotion-only rules it imposes on assignment statements, shown in Figure 2.5. If we wanted the integer part of a double value to be assigned to an integer variable, we would use casting. The following statements assign the value 1 to the integer variable `age`:

```
double daysSinceBirth = 401.5;
int age = (int) daysSinceBirth / 365;
```

The mixed mode arithmetic in the second statement produces the value 1.1, which the casting converts to an integer (1) before it is assigned to the variable age. If the casting were left out of the second statement, it would not translate because it would be a violation of the promotion rules given in Figure 2.5. This use of casting informs the translator that we are intentionally violating these rules.

2.6.4 The Math Class

If we were to examine the code of applications written by several different programmers, we would quickly come to the realization that mathematical calculations, such as raising a number to a power or calculating the square root of a number, are performed in many programs. For example, a program that computes the radius of a circle given its area would divide the area by the constant PI, and then take the square root of the result. Obviously, the accuracy of the calculation is dependent on a precise value of PI. In addition, because the square root is not one of the mathematical operators available in most programming languages, the programmer would have to know the algorithm for computing the square root of a number using the math operators available in the programming language.

To facilitate the coding of programs that use common mathematical constants and calculations, Java, like most programming languages, provides precoded libraries containing these constants and mathematical functions. The constants are coded as initialized variables, and the mathematical functions are coded into subprograms. In Java, subprograms are called *methods*, and related methods and variables are collected into **classes**. As discussed in Section 1.5.1, the collection of the precoded classes available in Java is called the Java Application Programmer Interface, or Java API. A complete description of the classes contained in the API, is available online. To locate this documentation, simply type "Java API Specification" into the search window of your browser.

The API class that contains mathematical constants and methods is called the Math class. Table 2.4 lists a mathematical constant and some of the most commonly used methods that are included in this class. The third column of the table gives a series of assignment statements that illustrate the use of the class's constants and methods. Notice that the name of the Math class followed by a dot precedes the name of the constant or method used in the statement. The angles used in the trigonometric functions that appear in the last three rows must be expressed in radians. The methods compute and return a value, which the coding examples in the rightmost column of the table assign to a variable.

Table 2.4
Commonly Used Math Class Constants and Methods

Constant or Method	Description	Coding Example
`PI`	The ratio of the circumference of a circle to its diameter (a `double`)	area = Math.PI * r * r;
`abs`	Computes and returns the absolute value of a number, n (returns the type it is sent)	nAbsolute = Math.abs(n);
`pow`	Computes and returns a number, n, raised to the power p (returns a `double`)	nToTheP = Math.pow(n, p);
`sqrt`	Computes and returns the square root of a number, n (returns a `double`)	rootN = Math.sqrt(n);
`toRadians`	Converts an angle, a, in degrees to radians (returns a `double`)	aRads = Math.toRadians(a);
`sin`	Computes and returns the sine of an angle, aRads, specified in *radians*	sinA = Math.sin(aRads);
`cos`	Computes and returns the cosine of an angle, aRads, specified in *radians*	cosA = Math.cos(aRads);
`tan`	Computes and returns the tangent of an angle, aRads, specified in *radians*	tanA = Math.tan(aRads);

The following code fragment calculates and outputs the sine of 45 degrees and 2 raised to the third power:

```
double angle = 45.0;
double angleInRadians = Math.toRadians(angle); //returns a double
double sineOfAngle = Math.sin(angleInRadians); //returns a double

System.out.println("The sine of " + angle + " is " + sineOfAngle);
System.out.println("2 cubed = " + Math.pow(2, 3));
```

Random Numbers

"A random number is a number generated by a process whose outcome is unpredictable and which cannot be subsequently reliably reproduced."[2] Random numbers are used in many computer applications such as game programs, encryption programs, and simulation programs. For example, flight simulator programs used to train pilots to react to air turbulence introduce turbulence into the flight at random times during the simulation.

The Math class contains a method named `random` that can be used to generate pseudorandom numbers. The numbers are not truly random because the sequence of numbers the method generates is based on the computer's real-time clock (i.e., the time of day) resolved to one millisecond, and therefore can be reliably reproduced.

The method returns a double in the range: 0.0 ≤ randomNumber < 1.0. (The highest number generated by the method is always less than 1.0.) The following code fragment outputs two random numbers in that range. The specific numbers output would depend on the time of day the code fragment was executed.

```
double randomNumber;
randomNumber = Math.random();
System.out.println(randomNumber);
randomNumber = Math.random();
System.out.println(randomNumber);
```

The method can be used to generate numbers in the range: min ≤ randomNumber < max using the assignment statements:

```
double randomNumber1 = min + Math.random() * (max - min);
int randomNumber2 = int(min + Math.random() * (max - min));
```

The second assignment statement uses casting to change the computed real number into an integer.

2.7 DIALOG BOX OUTPUT AND INPUT

Dialog boxes are a graphical way to communicate with the user of a Java program and offer an alternative to the console-based output produced by the `println` method in the System class. The *message dialog box* (Figure 2.6) is used to convey output to the user, and the **input dialog box** (Figure 2.7) is used to obtain input from the user. They are predefined graphical objects that automatically resize themselves to display the string argument sent to them. The string sent to a message dialog box is the text to be output to the user. In the case of an input dialog box, the string is an input prompt to be displayed to the user.

After a dialog box is displayed, the program execution is halted until the user clicks a button displayed in the box or strikes the return key. In the case of a graphics application, dialog boxes are normally used for all communication between the program and its user. Two methods in the class `JOptionPane`, `showMessageDialog`, and `showInputDialog` are used to display message (output) and input dialog boxes, respectively.

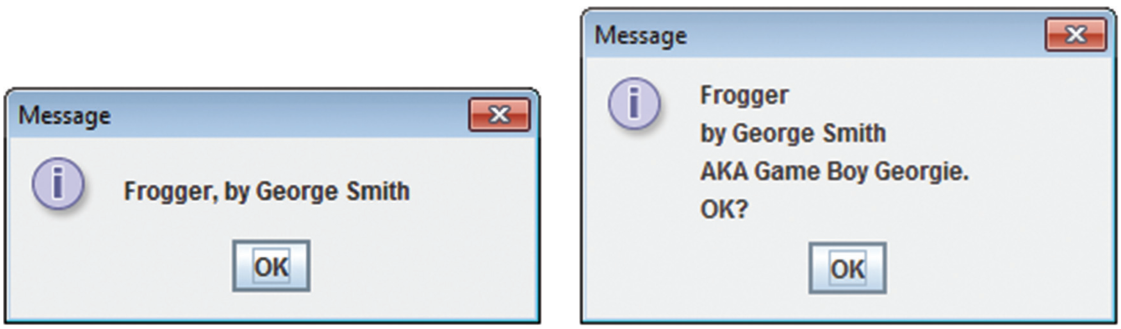
2.7.1 Message Dialog Boxes

The method `showMessageDialog` is a static method, as are the `Math` class's methods presented in Table 2.4 and its `random` method. As we will learn in Chapter 3, not all methods are static methods. When static methods are invoked, we must precede the name of the method with the name of its class followed by a dot. The `showMessageDialog` method and the `Math` class's `pow` method have another thing in common: they are both sent two arguments that are coded inside the parenthesis that follows the name of the method. For the `pow` method, we learned that these are the numbers to be raised to a power followed by the power to which to raise it.

In the case of the `showMessageDialog`, its two parameters describe the window to which the message box will be output followed by a string that contains the text of the output message. To output the message "Frogger, by George Smith," we would code

```
JOptionPane.showMessageDialog(null, "Frogger, by George Smith");
```

This would produce the message box shown in Figure 2.6(a). Coding `null` as the first argument causes the message dialog box to be displayed in the center of the monitor.



(a) (b)

Figure 2.6
Two message dialog boxes.

The second argument, the string, can contain all of the elements and features of the string sent to the `println` method used to perform output to the console. As we have learned, the string can be a concatenation of a mix of string literals and numeric variables. Just as with console output, the string will be output on one line unless new-line escape sequences (`\n`) are included in the string. The width and height of the message box will expand to accommodate the string. For example, the statement

```
JOptionPane.showMessageDialog(null, "Frogger" + "\nby George Smith," +
                                    "\nAKA Game Boy Georgie." + "\nOK?");
```

produces the output in Figure 2.6(b).

These features are especially useful in a very common use of a message box: to display a game's splash screen. A game splash screen is used to describe a game, its objective, and the manner in which the game pieces are controlled by the player. Usually, the name of the game and its creator (e.g., "Created by Game Boy Georgie") are also included.

2.7.2 Input Dialog Boxes

Input dialog boxes, which are displayed by the static method `showInputDialog`, are used to obtain input from the program's user (Figure 2.7). It is sent one argument, which the method displays as a prompt to the user. A text box is displayed below the prompt into which the user types the input. The box contains two buttons labeled "OK" and "Cancel." The string sent to the method can be a concatenation of a mix of string literals and variables, and the width and height of the input box is adjusted to accommodate the string and its embedded new-line escape sequences. Figure 2.7 shows the input dialog box produced by the statement

```
String s = JOptionPane.showInputDialog("Frogger has five " +
                                       "difficulty levels:" +
                                       "\n1 is the easiest," +
                                       "\n5 the most difficult," +
                                       "\nEnter your level");
```

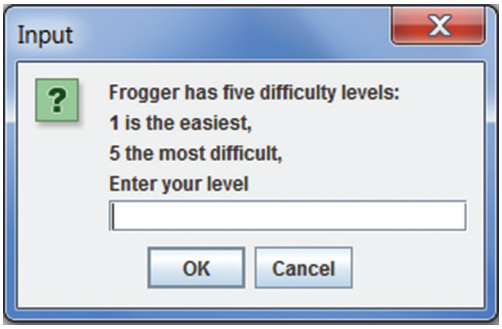


Figure 2.7
An input dialog box before the user enters input.

If, in response to the displayed prompt, the user types into the text box and then clicks "OK" (or strikes the `Enter` key), the location of a `String` object that stores the user input text would be placed in the reference variable `s`. (For brevity, we would say that the `InputDialogBox` method "returns a string," when in fact it actually creates and returns the address of a `String` object.) If the user clicked "OK" or struck the `Enter` key without making an entry in the text box, the returned `String` object would contain the empty string (""). Finally, if the user clicked "Cancel," `s` would store the null string. (It would be set to **`null`**.)

2.7.3 Parsing Strings into Numerics

Most of us would agree that there is a fundamental difference between the string `"one hundred seventy-six`" and the number 176. For one thing, we would not try to add the string "one hundred seventy six" to the string "ten" to obtain "one hundred eighty six". Rather, if we read the question, "what is the sum of one hundred seventy-six and ten?" we would first convert the numbers to their numeric representations, 176 and 10, and then perform the addition. In computer science, the difference between strings and numerics goes deeper than that because even if we were told to add "176" and "10," we would still have to convert these two strings to their numeric representations before performing the addition.

! TIP Operands in arithmetic expressions cannot be strings.

As discussed in Chapter 1, characters are stored using their Extended ASCII representation, and numerics are represented using their binary representation. Inside of String objects, the Extended ASCII representation is used. The difference between these two representations is shown below.

Extended ASCII Representation of 176	**Binary Representation of 176**
00110001 00110111 00110110	10110000
'1' '7' '6'	176

Because an input dialog box returns a string, when 176 is typed into its text box it returns the string "176", which must be converted to a numeric if it is to be used in an arithmetic expression. This conversion process is referred to as *parsing* strings into numerics. There is a set of classes in the API, called wrapper classes, which contain static methods to perform this conversion. The string to be converted to a numeric is sent to the method as an argument coded inside the open and close parentheses that follow the name of the method. The decision as to which class and method to use is based on the primitive numeric type the string is being converted to, as shown in Table 2.5.

Table 2.5
Numeric Wrapper Classes and Their Parsing Methods

To Convert a String to the Numeric Type	Use the Static Method	In the Wrapper Class
byte	parseByte	Byte
short	parseShort	Short
int	parseInt	Integer
long	parseLong	Long
float	parseFloat	Float
double	parseDouble	Double

To change the string literal "176" to its integer numeric representation, we would code:

```
int numericValue = Integer.parseInt("176");
```

To convert the string s to a numeric double, we would code:

```
double numericValue = Double.parseDouble(s);
```

Most often, the statements to accept a user input via an input dialog box, and the conversion of the returned string to a numeric, are coded one after the other.

```
String s =  JOPtionPane.showInputDiaog("Enter your age");
int age = Integer.parseInt(s);
s =  JOPtionPane.showInputDiaog("Enter your weight");
double weight = Double.parseDouble(s);
```

If the string sent to the wrapper class methods contains anything other than digits (i.e., the characters ‘0’, ‘1’, …, ‘9’), a *runtime error* `NumberFormatException` occurs, and the program terminates. If the empty string is passed to the methods (the user clicked “OK” in an input dialog box without typing an input) or the null string is passed to methods (the user clicked “Cancel” without making an entry), the same runtime error occurs. We will learn how to deal with these errors at runtime to bring the program to a more informative conclusion in Chapter 4 and how to permit the user to correct the erroneous input in Chapter 5 “Repeating Statements: Loops.”

NOTE *A runtime error is an error that occurs while the program is in execution.*

The application shown in Figure 2.8 calculates the area of a circle given its radius, and it also calculates the radius of a circle given its area. The inputs to the program (a radius of 10 and an area

```
import javax.swing.JOptionPane;

public class AssignmentMathAndDialogIO
{
   public static void main(String[] args)
   {
      String s;
      double area, radius;

      JOptionPane.showMessageDialog(null, "Circle area and radius" +
                                          "\n calculation program");
      s = JOptionPane.showInputDialog("To calculate an area," +
                                      "\n     enter a radius");
      radius = Double.parseDouble(s);
      area = Math.PI * Math.pow(radius, 2);
      JOptionPane.showMessageDialog(null, "The area of a circle" +
                                          " whose radius = " +
                                          radius + "\n is " + area);

      s =JOptionPane.showInputDialog("To calculate a radius" +
                                     "\n    enter an area");
      area = Double.parseDouble(s);
      radius = Math.sqrt(area / Math.PI);
      JOptionPane.showMessageDialog(null, "The radius of a circle" +
                                          " whose area = " +
                                          area + "\nis " + radius);
   }
}
```

Figure 2.8
The application `AssignmentMathAndDialogIO`.

of 200) and the corresponding outputs are show in Figure 2.9. The program demonstrates the use of assignment statements, parsing a string into a numeric, performing calculations, the use of the Math class, and dialog box I/O.

Line 1 of Figure 2.8 is an `import` statement. Import statements make API classes, and the constants and methods they contain, available to our programs. In this case, the class `JOption-Pane`, which contains the methods to perform dialog box input and output, is imported into the program. These methods are used to output the program's splash screen (lines 10–11) and to input the radius of a circle (lines 12–13). The new-line escape sequences (`\n`) in the strings sent to these methods produce a two-line message on the splash screen and a two-line input prompt, as shown at the top of Figure 2.9.

Line 14 parses the string representation of the input radius returned from the input dialog box into a double and assigns that double to the variable `radius`. Line 15 calculates the area of the circle. It uses the Math class's method `pow` to square the input radius and then multiplies that by the constant pi (`Math.PI`). Lines 16–18 output the radius and the computed area to a message dialog box (Figure 2.9c). The input radius and the calculated area are added to the output string with the use of the concatenation operator on line 18.

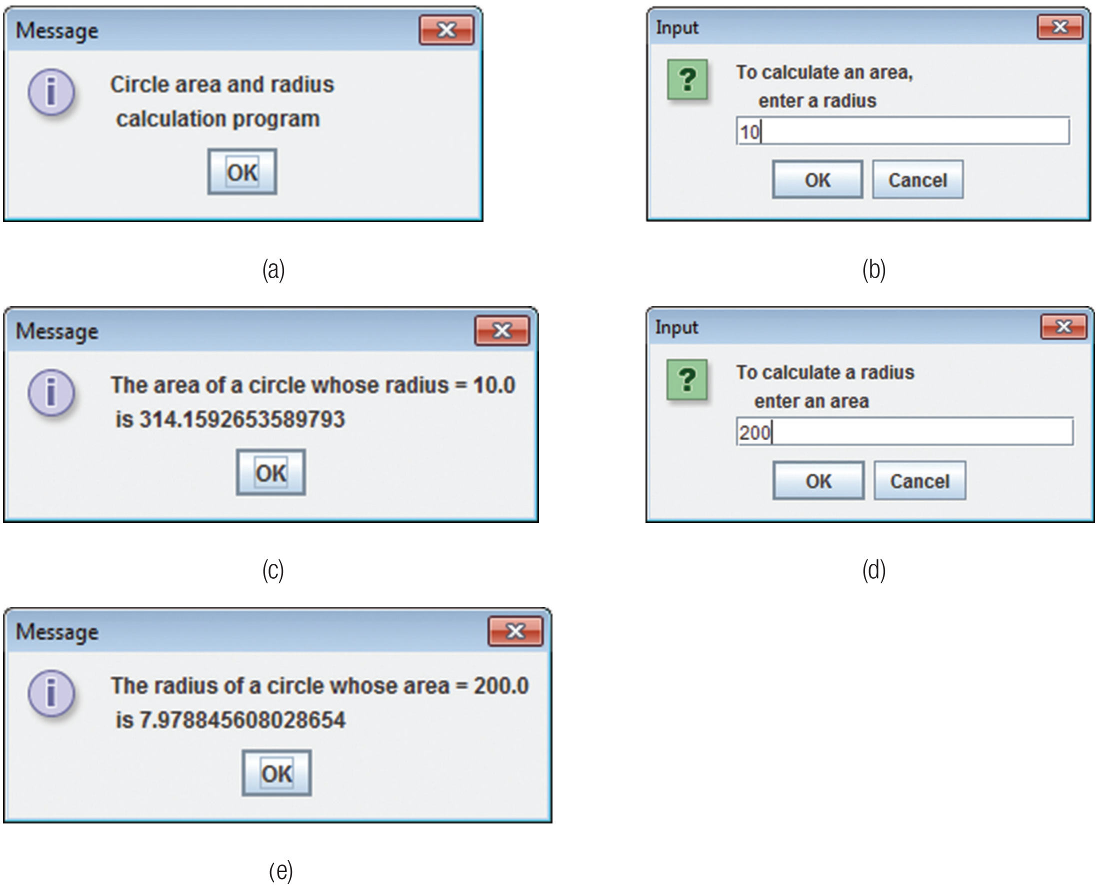


Figure 2.9
Input and resulting output from the application **`AssignmentMathAndDialogIO`**.

In a similar way, lines 20–26 accept an input area and compute the circle's radius. This calculation uses the Math class's `sqrt` method on line 23 to perform the calculation. The method accepts one argument, which in this case is the result of dividing the area by pi. The input to and output from this portion of the program is shown in Figure 2.9.

2.8 GRAPHICAL TEXT OUTPUT

In Section 2.4, we invoked (or some would say "used" or "called") the `println` and `print` methods of the `PrintStream` class to perform text output to the system console. In this section, we will learn how to use the method `drawString` in the API `Graphics` class to perform text output to a graphics window. This type of output is called graphical text output. Unlike console output, we can specify the font type, size, and style (e.g., bold style) of graphical text output. In addition, we can output the text to any location in a graphics window. In game programming, text output is typically used to display the game's level of difficulty, the remaining time, the score, or other information on the game's status.

With this added capability come added responsibilities. For example, every time a graphics window that has been minimized is restored, the graphical text must be output again, or it will not be visible in the window. In fact, anything that appears in the window before it was minimized must be redrawn when the window is restored. One mechanism for doing this in graphics programs is to place the graphical output in a *call back* method. Our game environment contains several call back methods. In this section, we will learn how to use the call back method `draw`, the `Graphic` class's `drawString` method, and how to set the font type, size, and style of graphical text output.

2.8.1 The `drawString` Method

The `drawString` method is a part (member) of the API Graphics class and is used to output text to a graphical object (perhaps a window). When the method is invoked three arguments are passed to it. The first argument is the text to be output. The second and third arguments are the x and y coordinates where the text will be output. These coordinates locate the lower left position of the first character of text. Their origin is the upper left corner of the graphical object in which the text is to be displayed (e.g., our game board), with x positive to the right and y positive down.

To output the text "Hello World" to our game environment's game board, positioned with the lower left corner of the "H" at (200, 300), we would code:

```
g.drawString("Hello World", 200, 300);
```

Notice that the two characters `g.` precede the name of the method. This is because the method must draw its text on a `Graphics` object. In our case, the object `g` would have to be a `Graphics` object attached to our game board because we want the text to be drawn on the game board. As we will see in the next section, the attachment of the object `g` to our game board is performed for us by the game environment.

A method that operates on an object is called a nonstatic method. The syntax used to invoke these methods is the name of the object it is operating on, followed by a dot, followed by the name of the method and its argument list. We have used this syntax in Section 2.4 to invoke the `print` and `println` methods. They were invoked by proceeding their names with the Java pre-defined `PrintStream` object `System.out` followed by a dot. (As previously discussed, when we invoke static methods, we precede the method's name with the name of the method's class followed by a dot. For example, `Math.sqrt(9);`)

The simplest way to determine if a method is static or nonstatic is to click on its class name in the lower left window of the online Java API Specification then scroll down through the API documentation to the method's name. If the method is static, the word "static" will appear in the column to the left of the methods name. For example:

`static double`	`sqrt(double a)` Returns the correctly rounded positive square root of a `double` value

If the method is a nonstatic method, the word "static" will not appear in the column to the left of the methods name.

The remaining issue is determining where in our game application program we place the invocation to `drawString`. The short answer is in the `draw` call back method, which we will discuss next.

2.8.2 The draw Call Back Method

Figure 2.10 presents the Java application class `GraphicalTextOutput` that creates a game window object on line 7, which is displayed by line 11 when the `main` method executes. Lines 2–12 are identical to that of the game code template in Figure 1.31 except for the change in the application's class name (lines 4 and 6) and the title of the window (line 7). When the program is run, the window shown in Figure 2.11 appears on the monitor.

```
1   import edu.sjcny.gpv1.*;
2   import java.awt.Graphics;
3
4   public class GraphicalTextOutput extends DrawableAdapter
5   {
6     static GraphicalTextOutput ga = new GraphicalTextOutput( );
7     static GameBoard gb = new GameBoard(ga, "Graphical Text Output");
8
9     public static void main(String[] args)
10    {
11      showGameBoard(gb);
12    }
13
14    public void draw(Graphics g) //the drawing call back method
15    {
16      g.drawString("Hello World", 250, 220);
17
18    }
19  }
```

Figure 2.10
The application `GraphicalTextOutput.`

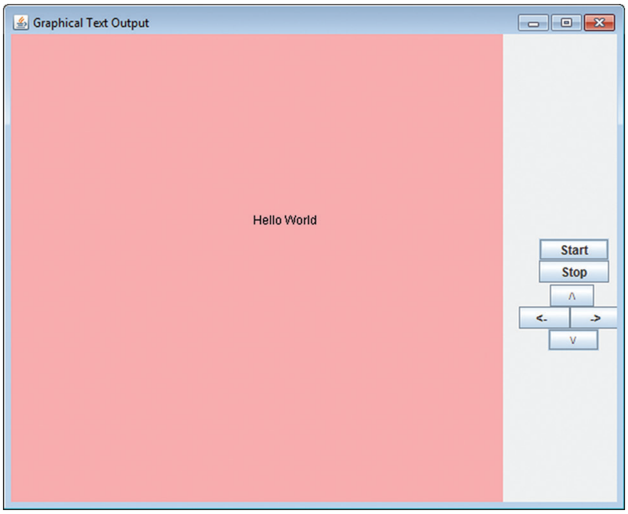


Figure 2.11
The output produced by the application GraphicalTextOutput.

Lines 14–18 is a coding of the game environment's `draw` call back method. The first line of the `draw` method, called the method's signature, must always be identical to the code on line 14, and it requires that the game environment be added (imported) to the program (line 1). The invocation to the `drawString` method has been coded on line 16, which, when executed, outputs the graphical text "Hello World" to our game board beginning at pixel location (250, 220). To use the `drawString` method the Graphics class must be imported into the program (line 2). But, when does it execute?

We have learned that the `main` method is invoked by the Java runtime environment causing its statements to execute, beginning with its first executable statement (the program entry point) and ending when the execution reaches the end of its code block, in this case, line 12. Thus, it would appear that line 11 would display the application's window, and then the program would end. But the `draw` method must have executed because the characters *Hello World* appear in the program's window (Figure 2.11). So again, we ask the question, "when does it execute?"

The answer is fundamental to why the method `draw` is referred to as a call back method. Line 11 in the main method invokes the method `showGameBoard`, which requests that the game environment display the game board window. Before the game environment displays the game's window, it invokes, or calls back, the `draw` method coded in the application, causing it to execute. When the `draw` method ends, the game environment completes the display of the game's window requested by the `showGameBoard` method. Thus, the application calls the game environment to display the game board window (line 11), and the game environment calls back the application's `draw` method to perform its drawings on the game board before the window is displayed. Specifically, the execution sequence is line 11 in the method `main`, the code of the `showGameBoard` method, the code at the beginning of a method in the game environment, the `draw` call back method lines 14-18), and finally, the remainder of the code in the game-environment method.

In fact, every time the program's window has to be redrawn (e.g., the window was minimized and then the window's icon on the status bar is clicked), the game environment's code invokes the `draw` method to redo its drawings on the game board. This can easily be verified by adding the statement

```
System.out.println("the draw method was invoked");
```

to the `draw` call back method. Then, every time the `draw` method is invoked, we will see an output on the system console.

NOTE *Even though the main method containing the program's entry point ends its execution, a graphical program continues to execute until the program's graphical window (e.g., the gameboard window) is closed.*

2.8.3 The setFont Method: A First Look

Like the `drawString` method, the `setFont` method is a part (member) of the API `Graphics` class. It is used to change font type, style, and size. The output in Figure 2.11 used the default font values. Once changed, all subsequent graphical text output will use the new (or current) font type, style, and size until it is changed again. The method is passed one argument. The following code, when added to the `draw` method, changes the font type to Arial, the style to bold italic, and the font size to 16 points, and then outputs the text *The Font was Changed*. The syntax of the argument sent to the `setFont` method will be explained in Chapter 3.

```
g.setFont(new Font("Arial", Font.BOLD + Font.ITALIC, 16));
g.drawString("The Font was Changed", 150, 300);
```

2.9 THE COUNTING ALGORITHM

Counting is something that is done in most programs and is considered to be a fundamental algorithm in computer science. For example, in game programs it is used to count the number of seconds remaining in a game or the number of seconds since the game began. In the first case, the time starts at a designated amount of time and counts down to zero; in the second case, the time starts at zero and counts up. In both cases, the game's time is usually displayed on the game board. In this section, we will discuss the counting algorithm, and we will learn how to use it inside the game environment's call back method `timer1` to count seconds.

Most of us began to learn how to count by memorizing the integers beginning with 1. Our parents may have said to us, "say this: one, two, three, four." Most of us, on the first try, perhaps said "three, four," or "one, two, four," or some other erroneous sequence. Through repetition, eventually we memorized the sequence and extended it by recognizing that each new element is "one more."

Somewhere along our cognitive development path, we discovered the counting algorithm. In support of that is the realization that most people never memorized the integers from 1,242,518 to 1,243,589. However, most of us could recite that sequence of integers if asked to do so because we use the counting algorithm to determine the sequence. Below is the generalized counting algorithm that can be used to count forward or backward by any increment:

```
int count = aBeginningValue;
// repeat the next statement until count reaches the ending value
count = count + aCountingIncrement
```

For example, to count upward from 1 to 10 by 1s, we code:

```
int count = 1;
// repeat the next statement until count reaches 10
count = count + 1; // 1 becomes 2, 2 becomes 3, 3 becomes 4, …
```

To count backward by 5s, from 1,165 to 875, we code:

```
int count = 1165;
// repeat the next statement until count reaches 875
count = count  + -5; // 1165 becomes 1160, 1160 becomes 1155, …
```

Repeating statements is the topic of Chapter 5, so we will revisit the counting algorithm in that chapter. However, if we want to count seconds within a game program, the second line of the counting algorithm can be repeated by placing it inside a call back method named `timer1`. This method is invoked by the game environment once every second causing the statement to be repeated once a second.

2.9.1 A Counting Application: Displaying a Game's Time

The game environment has three timer call back methods named `timer1`, `timer2`, and `timer3`. Their signatures (first lines) are:

```
public void timer1( )
public void timer2( )
public void timer3( )
```

If you code these methods into your game program, they will be invoked every time their respective timers "tick." For example, the method `timer2` will be invoked every time `timer2` ticks. Because counting seconds is so common in games, by default `timer1` ticks every second. It begins ticking when the game window's `Start` button is clicked, pauses when the `Stop` button is clicked, and resumes ticking when the `Start` button is clicked. After a timer call back method ends its execution, the game environment invokes the `draw` call back method. The details of the other two timers, which are normally used to animate game objects, will be discussed in Chapter 6.

Figure 2.12 presents the graphical application `CountingSeconds` that illustrates the use of the counting algorithm to count upwards by one, starting from zero. The output produced by the program three seconds after the user clicks the `Start` button is shown in Figure 2.13.

The declaration of the counter variable `count` and its initialization to zero seconds is coded on line 8. (Note that the key word `static` is coded at the beginning of this line. The need for it will be explained in Chapter 3.) Declaring this variable on line 8 places it outside of the code blocks (the open and close braces) of all of the class's methods, which makes it makes available to *all* of the class's methods. Variables declared in this way are said to be *class level* variables.

```
import edu.sjcny.gpv1.*;
import java.awt.Graphics;
import java.awt.Font;
public class CountingSeconds extends DrawableAdapter
{
  static CountingSeconds ga = new CountingSeconds( );
  static GameBoard gb = new GameBoard(ga, "The Counting Algorithm");
  static int count = 0;   // a class level variable

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }

  public void draw(Graphics g) // the drawing call back method
  {
    g.setFont(new Font("Arial", Font.BOLD, 18));
    g.drawString("Your game time is: " + count, 10, 50);
  }

  public void timer1()
  {
    count = count + 1;
  }
}
```

Figure 2.12
The application `CountingSeconds.`

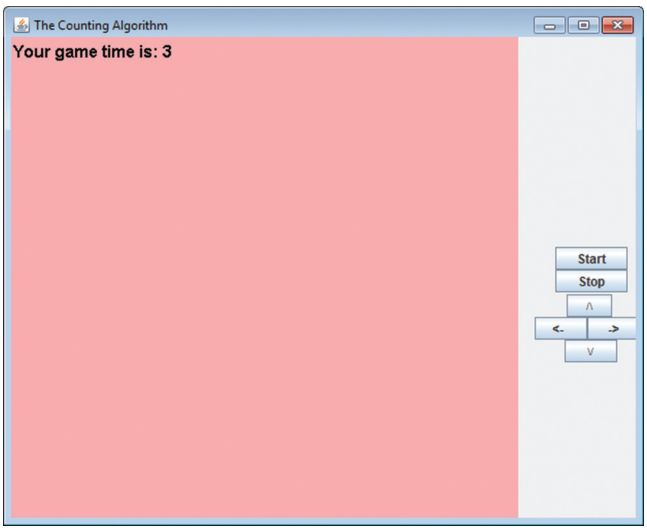


Figure 2.13
The output produced by the application CountingSeconds three seconds after the Start button is clicked.

The second line of the counting algorithm is coded on line 23 inside the `timer1` call back method (lines 21–24). Because timer1 ticks once a second (by default), line 23 is repeated every second, causing the counter variable `count` to count seconds. Each time the `timer1` method ends, the game environment invokes the `draw` method, which displays the new time on the game board by outputting the contents of the class-level variable `count` (line 18). The output appears in bold Arial 18 point font because line 17 invokes the *Graphic* class's `setFont` method to change the current font values. The import statement on line 3 makes the constants and methods of the *Font* class available for use by that statement.

2.10 FORMATTING NUMERIC OUTPUT: A FIRST PASS

In this chapter, we have discussed how to perform numeric output, but we have not discussed how to format numeric output to improve its readability, such as adding a comma every three digits to the left of the decimal point, or adjusting the precision of the fractional part of a number output to the right of the decimal point. The `format` method in the `DecimalFormat` class can be used to accomplish both of these commonly used types of output formatting. We will conclude this chapter with an introduction to the techniques used to format numeric output; we will present more details on these techniques in Chapter 5.

The application presented in Figure 2.14 uses the `DecimalFormat` class's `format` method to format the output of a real number (line 12) and an integer (line 13). The method returns a string containing the formatted numeric value and is sent one argument. The argument is the numeric variable to be formatted: `speedOfLight` and `population` on lines 12 and 13 respectively.

This nonstatic method is invoked (on lines 12 and 13) using the `DecimalFormat` object `df` declared on line 10. The formatting to be performed is associated with the decimal format object and is specified as a string literal coded inside the parentheses at the end of line 10. The string literal `#,###.##` coded on this line indicates that commas will appear every three digits to the left of

```
1   import java.text.DecimalFormat;
2
3   public class BasicNumericFormatting
4   {
5     public static void main(String[] args)
6     {
7       double speedOfLight = 299792458.7153;
8       int population = 1097603176;
9
10      DecimalFormat df = new DecimalFormat("#,###.##");
11
12      System.out.println(df.format(speedOfLight));
13      System.out.println(df.format(population));
14    }
15  }
```

Figure 2.14
The application `BasicNumericFormatting`.

the decimal point, and *real* numbers (nonintegers) will be output with two digits of precision. The level of precision used in the formatting can be changed by adding or removing pound signs to the right of the decimal point on line 10.

The output produced by the program is given in Figure 2.15. As this output shows, real numbers are always rounded up (299792458.7153 was output as 299792458.72), and integer output does not contain a decimal point.

```
299,792,458.72
1,097,603,176
```

Figure 2.15
The output produced by the `BasicNumericFormatting` application.

2.10.1 The `printf` Method

Another way of formatting numeric output to the System console is to use the `printf` method. Like the `print` and `println` methods it is invoked on `System.out`, but the information passed to it is not simply the items to be output. Rather it is passed two sets of information separated by a comma: the formatting information for each item to be output, followed by the items to be output.

`System.out.printf`("Formatting Information", Items To Be Output);

The items to be output are a sequence of variables and/or string literals each separated by a comma, normally followed by a space for readability purposes. The formatting information is a string that contains a sequence of format specifiers, one for each item to be output. Each specifier begins with a % sign followed by a description of the item's formatting. The % sign can be preceded by a sequence of characters to be output before an item is output.

Within the formatting information, there must be a format specifier for each item to be output. The format specifiers are associated with the items to be output in the order in which they are coded. For example, the third format specifier would be used to format the third item to be output.

Replacing lines 10-13 in Figure 2.14 with line 2 shown below would produce the first line of output shown in Figure 2.15. Replacing them with line 4 shown below would produce both lines of output shown in the figure. Lines 6-7 annotates the output as shown below line 7. Within the formatting information, an output's annotation is placed before the % signs, as shown on line 6.

```
1  // produces the first output shown in Figure 2.15
2  System.out.printf("%,.2f \n", speedOfLight);
3  // produces both outputs shown in Figure 2.15
4  System.out.printf("%,.2f \n%,d  \n", speedOfLight, population);
5  // produces the below output
6  System.out.printf("Light speed: %,.2f  \nPopulation: %,d  \n",
7                         speedOfLight, population);
Light speed: 299,792,458.72
Population: 1,097,603,176
```

Within the above formatting specifiers the comma after the % sign imbeds a comma within a numeric output every three digits, and the `.2` produces two digits of precision rounded up. The characters `f` and `d` are use to format the output as a floating point decimal or a signed decimal integer respectively. The upside of this approach to formatting numeric output is that it is more succinct, however it is also more cryptic and the ordering of the formatting text within the format specifiers, placed after the percent sign, is important.

A good use of the `printf` method is to align outputs. The below code produces the output shown below it. Within the format specifiers on line 3, the `14` specifies the width of the output item. The – (dash) indicates the output should be left justified within the output width. The default is right justified. The `s` produces string output.

```
1  String[] paint = {"Paint", "Pan and Roller"};
2  String[] screws = {"Philips head", "Slot head"};
3  System.out.printf("%-14s %-14s \n%-14s %-14s \n",
4                    paint[0], paint[1], screws[0], screws[1]);
```

Output Produced

Paint	Pan and Roller
Philips head	Slot head

2.11 CHAPTER SUMMARY

This chapter introduced the basic components of a Java program and the Java template for developing a program that performs input, mathematical calculations, and output. Variables are declared to store data during the program's execution. Primitive variables store one data value, and reference variables store a memory address where the object that contains the data is located. The type of the data (for example, character or integer), must also be declared. Good coding style dictates that variable names be meaningful as well as syntactically correct. Meaningful variable names indicate what the data represents. They begin with a letter and cannot contain spaces.

The `print` and `println` methods are used to output a string to the console window to which the object `System.out` is attached. Escape sequences permit characters with special meaning to be used in output statements. The concatenation operator joins data values and strings into a single output string.

`String` objects, which are reference variables store the memory address that refers to, or references, the actual string. Strings can be created and initialized using either a one-line or two-line grammar. The default value for an uninitialized string is `null`. Strings have to be converted into numeric values to perform mathematical operations, and there is a set of classes in the API, called wrapper classes, which contain static methods to perform this conversion.

Java, like most programming languages, provides the ability to perform basic arithmetic calculations and provides additional features, including the API `Math` class, to perform more complex calculations. Arithmetic calculations are performed in Java using arithmetic expressions. Arithmetic expressions consist of a series of operands separated by operators. The parts of an arithmetic expression inside the parentheses are evaluated first using the rules of precedence to determine the order of the operations. If the operators are of equal precedence, they are evaluated from left to right. Higher precedence operators are evaluated first. The division of two integers always results in a truncated integer value, and the mod operator is used to determine the remainder of integer division.

Values are assigned to variables using the assignment operator. Generally, the type of the value being assigned to a variable should match the type of the variable. Type casting and promotion are provided and are used with mixed-mode expressions to ensure that the variable types are compatible and there is no loss of precision.

Dialog boxes are a graphical way to communicate with the user of a Java program and offer an alternative to console-based input and output. The method `drawString` in the API `Graphics` class can be used to perform text output to a graphics window, and the setFont method can be used to change the default font, style and size. The counting algorithm is used to keep track of elapsed seconds in a game program. Chapter 3 will extend the concepts of objects, classes, and methods and their application to creating game programs.

Finally, the `DecimalFormat` class is introduced to provide ways to format numeric output to improve its readability.

Knowledge Exercises

1. True or False:
   a) The type of the data stored in a variable can change as the program executes.
   b) Variables must be declared in a program before they are used.
   c) Variables must be initialized when they are declared.
   d) It is grammatically incorrect to begin a variable name with an upper-case letter.
   e) Spaces can be used in variable names for better readability.

2. Which statement in a Java application program is executed first?

3. What are variables? Name two types of variables and the information each one stores.

4. Which of the following is not a primitive data type?
   a) boolean **b)** char **c)** String **d)** int

5. Give the default initial values for variables declared to be of the following types:
   a) boolean **b)** char **c)** double **d)** int

6. Write a well-composed declaration statement to declare variables that can store:
   a) Maggie's age initialized to 32
   b) The first initial of Ryan's name initialized to the letter 'R'
   c) The cost of a taco
   d) The number 21,234,096,464
   e) The fact that it is snowing

7. Is a numeric literal, coded in a program, represented as a float or a double? Explain.

8. Determine if each of the following variables is well composed and valid. For those that are not, explain why not.

   a) 2ndplace　**b)** middleInitial　**c)** winningTeam
   d) fgp3　**e)** test1 grade　**f)** myScore
   g) SalePrice

9. Write the code to output two lines to the system console. The first line will contain your name, and the second line will contain the town in which you live, using:

   a) Two `println` statements　**b)** One `println` statement

   Then give the equivalent `printf` statements.

10. Write a well-composed variable declaration statement to declare a string String object initialized to Skyler's address, which is 21 First Avenue, using the:

    a) One-line object-declaration grammar　**b)** Two-line object-declaration grammar

11. Draw a picture of the memory allocated by the statements:

    a) `int distance = 675;`　**b)** `String myName = new String("Jane");`

12. Give one statement to:

    a) Output the annotated contents of the memory cell `priceOfCorn`
    b) Output the sentence: Martin said: "I had a dream."
    c) Change the contents of the variable `myBalance` to 234.54.

13. Evaluate each of these expressions:

    a) 17 - 5 * 2 + 12　**b)** 31 - 7 * 2 + 14
    c) (48 + 12) / 12 + 18 * 2　**d)** 21 - 9 + 18 + 4 * 3.7

14. Give the code to:

    a) Declare the variable `quizAverage` and store the average of the variables `quiz1`, `quiz2`, `quiz3`, and `quiz4` in it
    b) Calculate the sine of 45 degrees and store the value in the variable `sineOf45`
    c) Calculate and output the square root of 45.67
    d) Calculate and output 34.7 to the 5^{th} power

15. Write the variable declaration to declare the variable `average` and the assignment statement to store the average of three speed limits: 55, 57, and 60 miles per hour.

16. Give an assignment statement to store the integer part of the value stored in the double variable `bankBalance` in the variable `dollars`.

17. True or False:

    a) You must include an import statement in a program to perform I/O using dialog boxes.
    b) A message dialog box can be used to obtain input from the program user.

c) A string is always returned from an input dialog box.

d) When the user clicks “OK” without making an input into an input dialog box, null is returned.

e) Dialog boxes will size themselves to accommodate the string argument sent to them.

18. Write the code to output two lines to a message dialog box. The first line will contain your first and last name, and the second line will contain your date of birth in the format “My birthday is: dd\mm\yyyy” (yes, those are backslashes).

19. Give the code to allow the program user to enter a checking account balance using an input dialog box. Include a well-composed user prompt.

20. Think of a game. Write the code to output the name of the game and its creator, the task (objective) of the game, and how the game pieces are controlled to a message dialog box of reasonable size.

21. Give the code to declare a double variable named `deposit` and to parse the input contained in the string `sDeposit` into it.

22. Write the code to declare an integer variable named `speedLimit` and to parse the input contained in the string `sSpeedLimit` into it.

Programming Exercises

1. Write a Java application that outputs your name on one line followed by the town in which you live to the system console.

2. Write a program to calculate the average of five quiz grades: 100, 97, 67, 85, and 79. Output the quiz grades and the average to the system console. The output should be well annotated with the quiz grades on one line and the average on another.

3. Write a program to accept an angle (input in degrees) and a real number. Then, output the angle and its sine, cosine, and tangent. Follow that output with the output of the input real number, its cube, and the square root of the number. The outputs should occupy several lines and be sent to both the system console and to a message dialog box. The input prompts should be well composed, and the outputs should be well annotated.

4. Repeat Programming Exercise 3, but output the information to the system console and to the middle of the game board. Use 20-point italic Arial font for the game-board output.

5. Write a program to ask the user to enter the product of a pair of real numbers (of your choosing), with the input rounded to one digit of precision. After the product is entered, output the user’s input and the correct product, rounded to one digit of precision. The outputs should occupy several lines, and be sent to the system console and to a message dialog box. The input prompts should be well composed, and the outputs should be well annotated.

6. Write a program to ask the user to enter the product of a pair of real numbers of your choosing. After the product is entered, output the correct answer and the number of seconds it took the user to enter the product to the center of the game board and to the system console. Use 20-point italic Arial font for the game-board output. The output should be on two lines and well annotated.

7. Repeat Programming Exercise 6, but output the numbers with commas every three digits on the left side of the decimal point, and use one digit of precision.

Endnotes

[1] The URL of the Edition 7 API documentation is: *http://docs.oracle.com/javase/7/docs/api/* It is named: Java Platform, Standard Edition 7 API Specification.

[2] http://www.randomnumbers.info/content/Random.htm

CHAPTER 3

METHODS, CLASSES, AND OBJECTS: A FIRST LOOK

In this chapter

This chapter extends the concepts of methods, classes, and objects discussed in the previous chapter to enable us to design and implement our own classes and the methods that they contain. These concepts facilitate the development of our programs by allowing us to divide a large program into several smaller classes, separately develop these classes, and then integrate them into the larger program. Once written, these classes can also be used in other programs, just as the API classes are. Several design tools will be introduced in the chapter to methodize the specification of a class and the object it defines. The understanding of material presented in this chapter is the foundation of the advanced OOP topics discussed in Chapters 7 and 8.

After successfully completing this chapter, you should:

- Be able to write void and nonvoid methods
- Understand how to share primitive information and objects between methods
- Understand the concept of value parameters
- Be able to read a Unified Modeling Language (UML) diagram and use it to specify a class
- Understand how to design and specify graphical and nongraphical objects
- Be able to identify, write, and use a set of methods that most classes contain
- Understand the concept and use of public and private data members and methods
- Be able to design, construct, modify, and access an object using its class's methods
- Use methods of the `Graphics` class to draw lines, rectangles, ovals, and circles
- Have acquired the foundational skills required for a study of Chapters 7 and 8

3.1 METHODS WE WRITE

In Chapter 2, we became familiar with several methods available in the Java Application Programmer Interface. For example, the methods `println` and `print` perform output to the system console, `pow` and `sqrt` perform calculations, and `drawString` and `setFont` perform text output to the program window. Being resident in the API, these methods are available to all Java programmers, and their use expedites the program development process because only one programmer, the API programmer, had to discover their algorithms and then write, test, and debug their code. The rest of us simply use the methods by importing them into our program and writing a one-line invocation statement. Because most of the cost of software development is the salaries paid to the programmers, the use of prewritten methods also makes software more affordable.

In this section, we will learn how to write our own methods. Not only will this allow us to re-use the code that we write in other programs, but it also facilitates the development of our programs by dividing a large program into several smaller subprograms called methods. By dividing a large program into subprograms, these methods can be developed by several programmers working in parallel, which greatly reduces the calendar time required to produce a program.

NOTE

The Motivation for Writing Methods
Extends our problem solving capabilities: Humans are good at solving small problems but not large problems
Reduces development time: Methods can be developed in parallel by several members of a programming team
Reduces cost: Methods can be written in such a way that they can be used in any program using a one-line invocation statement

3.1.1 Syntax of a Method

In Java, all the methods we write must be part of a class. They must be coded within the class's code block, the open and close braces that begin and end a class statement. The class statement can be the one that contains the program entry point, the method `main`, or some other class that we will learn how to create later in this chapter. When methods are coded inside the class that contains the method `main`, good coding style dictates that they be coded after it.

The minimum code required to create a method is:

```
returnType=methodName( )
{
   //the code of the method is placed here
}
```

The first line of the method is called the method's *signature*. The signature is followed by a set of open and close braces that define the bounds of the method's *code block*. The statements to be executed when the method is invoked are coded inside this code block.

The method's signature, the first line of the method's code, must include the type of the information returned from the method, followed by the method's name, followed by an open and close set of parentheses. If the method does not return a value to the invoker, for example, compute and return the square root of a number, the keyword **void** is used as the return type. In this case, the method is said to be a `void` method.

A method that simply outputs the name of the student newspaper to the system console every time it is invoked would be an example of a `void` method.

```
void outputNewspaperName()
{
   System.out.println("The Student Voice");
}
```

The syntax and coding style used for naming methods are the same as those used to name variables:

- they cannot contain spaces
- they should begin with a lower-case letter
- new words should begin with an upper-case letter

Normally, they are only comprised of letters. Digits, the dollar sign, and the underscore are not normally used in their names. For example, a method that adds two integers together and returns the result could be named `addTwoInts` rather than `add_2_Ints`.

Figure 3.1 presents the application `AVoidMethod` that contains the implementation and two different invocation forms of the void method `outputNewspaperName`. The output it produces is shown in Figure 3.2.

```
public class AVoidMethod
{
   public static void main(String[] args)
   {
      AVoidMethod.outputNewspaperName(); //1st method invocation
      System.out.println("Page 1\n");
      outputNewspaperName();             //2nd method invocation
      System.out.println("Page 2\n");
   }

   static void outputNewspaperName()     //method signature
   {
      System.out.println("The Student Voice");
   }
}
```

Figure 3.1
The console application **AVoidMethod**.

```
The Student Voice
Page 1

The Student Voice
Page 2
```

Figure 3.2
The output produced by the console application **AVoidMethod**.

The program consists of two methods: the method `main` (lines 3–9) and the method `output-NewspaperName` (lines 11–14). Both of these methods are coded within the program class's code block that begins on line 2 and ends on line 15.

NOTE *A method cannot be coded inside of another method's code block.*

The signature of the method `outputNewspaperName`, coded on line 11, begins with the key word **`static`**. Not all method signatures begin with this key word. As we have learned, methods fall into two categories: those that operate on objects (nonstatic methods) and those that do not (static methods). An example of a method that operates on an object is the method `println`. It operates on, or sends its output string to, the console object whose name is `System.out`. Methods that do not operate on an object must include the key word **`static`** in their signature. The method `outputNewspaperName` does not operate on an object, so its signature begins with the key word static. In Section 3.5, we will discuss methods that we write that do operate on objects, and we will gain more insight into what it means to say a method operates on an object.

NOTE *Methods that do not operate on an object must include the key word* **`static`** *in their signature.*

The method `outputNewspaperName` is invoked in lines 5 and 7 of the application's main method. Line 7 just mentions the name of the method followed by open and close parentheses. This invocation syntax is valid because the static method is coded within the same class, `AVoid-Method,` as the invocation statement (line 7). The more generalized syntax for invoking a static method is used on line 5. Here, the invocation statement begins with the name of the class in which the method is coded followed by a dot:

```
AVoidMethodApp.outputNewspaperName();
```

We used this syntax to invoke the static methods `pow` and `sqrt` that are coded in the Math class.

```
double ans = Math.pow(3.0, 2.0);
double root = Math.srt(9.0);
```

Because these two methods are not coded in the same class in which they are normally invoked, the name of the class must be included in the invocation statement. The only exception to this is the use of a static import statement. When either syntax is valid, the shorter syntax makes our programs more readable and is therefore preferred.

The execution sequence of the application begins on line 5 of the main method. This invocation statement causes the code in the code block of the `outputNewspaperName` method to execute (lines 12–14), which produces the first line of output (Figure 3.2). Then line 6 of the main method executes, producing the second line of output. Line 7 causes the `outputNewspaperName` method to execute a second time, which produces the third line of output. Finally, line 8 executes, which produces the last output line.

NOTE *After a method executes, the next statement to execute is the statement immediately after the statement that invoked it.*

3.2 INFORMATION PASSING

For a method to function properly, information must often be passed to it when it is invoked, and some methods must return one piece of information to the invoking statement. Consider the Math class's nonvoid static method `pow`. When it is invoked, a number (n) and a power (p) are passed to it, and the method returns the result of its calculation: n^p. The left side of Figure 3.3 depicts this sharing of information between the invoker (top left) of the method `pow` and the method (bottom left). The right side of the figure generalizes this concept of shared information between the invoker, often called the client, and the method that performs some "work" for the client, often referred to as the worker method. For example `pow`'s work is to compute a given number (n) raised to a given power (p).

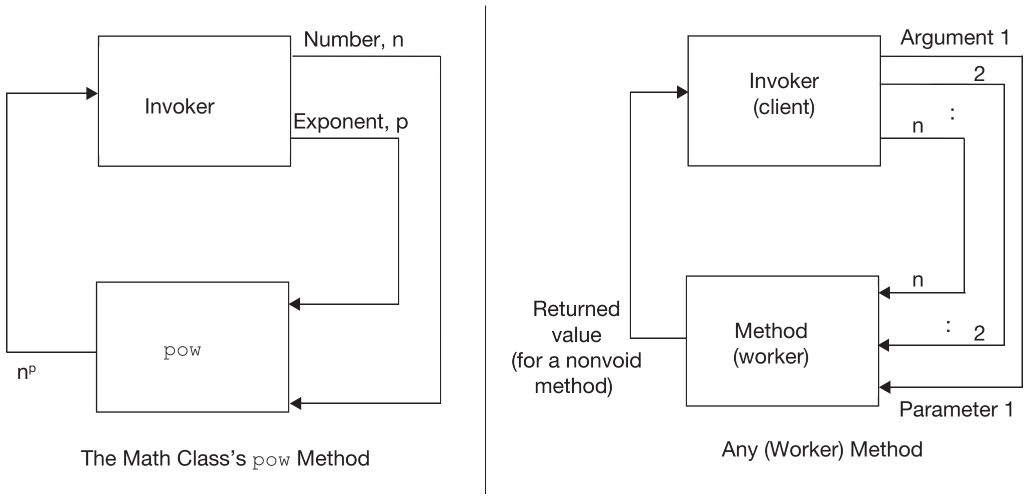


Figure 3.3
The sharing of information between a method and its invoker.

As depicted in the right side of Figure 3.3, an unlimited number of pieces of information can be sent from the invoking client to a worker method, however, only one piece of information can be sent back to the client from the worker method. We say that the client sends the method an *argument list* containing the shared information, and the worker method receives the shared information in its *parameter list* (either or both of which could be empty).

NOTE *No more than one piece of information can be returned from a method.*

3.2.1 Parameters and Arguments

If a method is to receive information passed to it from the client, then its signature must contain a parameter list. The parameter list is coded inside the open and close parenthesis of the method's signature. The parameters in the list are separated by commas, and each parameter consists of a variable name preceded by its type. For example, the signature of a method whose work is to output a person's age and weight would have an `int` and a `double` parameter in its parameter list.

```
static void outputAgeAndWeight(int age, double weight)
{
   System.out.println(age: "  + age + " weight: " + weight);
}
```

Each parameter receives *one* piece of information sent to it by the client code's invocation statement, and the *type* of the parameter must match the type of the information sent to it. The client's statement used to invoke the method `outputAgeAndWeight` would contain two arguments in its argument list, within parentheses.

```
outputAgeAndWeight(myAge, myWeight);
```

This statement passes the contents of the variables `myAge` and `myWeight` to the method.

Arguments, information that is to be shared with the worker method, can be variables (e.g., `myAge, myWeight`) that have been previously declared in the client code, or string or primitive literals.

NOTE *The order, number, and type of the arguments in a method invocation statement must match the order, number, and type of the parameters in the method's signature.*

Each time a method is invoked, the variables in the method's parameter list are allocated and paired up with the arguments in the invocation's argument list (the first parameter paired with the first argument, the second parameter paired with the second argument, etc.), and the value stored in each of the arguments is copied into the paired parameters. For the invocation statement

```
outputAgeAndWeight(myAge, myWeight);
```

the value in the argument `myAge` is copied into the parameter `age` of the method `outputAgeAndWeight`, and the value contained in the argument `myWeight` is copied into the parameter `weight`. This type of information passing is called *passing by value*, and the parameters are called *value parameters* because the values contained in the arguments are copied into the parameters. Once the parameters have been allocated and this transfer of information is complete, the code in the worker method's code block begins execution.

Consider the sequence of code that contains a main method and the method `outputAgeAndWeight`:

```
public static void main(String[] args)
{  int myAge = 23;
   double myWeight = 185.4;

   outputAgeAndWeight(myAge, myWeight);
}

static void outputAgeAndWeight(int age, double weight)
{
   System.out.println("age: "  + age + " weight: " + weight);
}
```

Figure 3.4 depicts a sequence of seven events that occur when this code executes and illustrates the process of passing information using value parameters. The left side of figure shows the main method's (client) code and its execution sequence (events 1, 2, 3), which includes the RAM memory allocated to its two arguments (event 2 is depicted in the bottom left portion of the figure).

The right side of the figure shows the code of the method and its execution sequence (events 4, 5, 6, and 7), which includes the RAM memory allocated to its two parameters (event 5 is depicted in the bottom right portion of the figure). The passing of the values stored in the client's arguments into the paired worker parameters is depicted as event 6 in the bottom center of the figure. After the information is passed, the code block of the method executes (event 7).

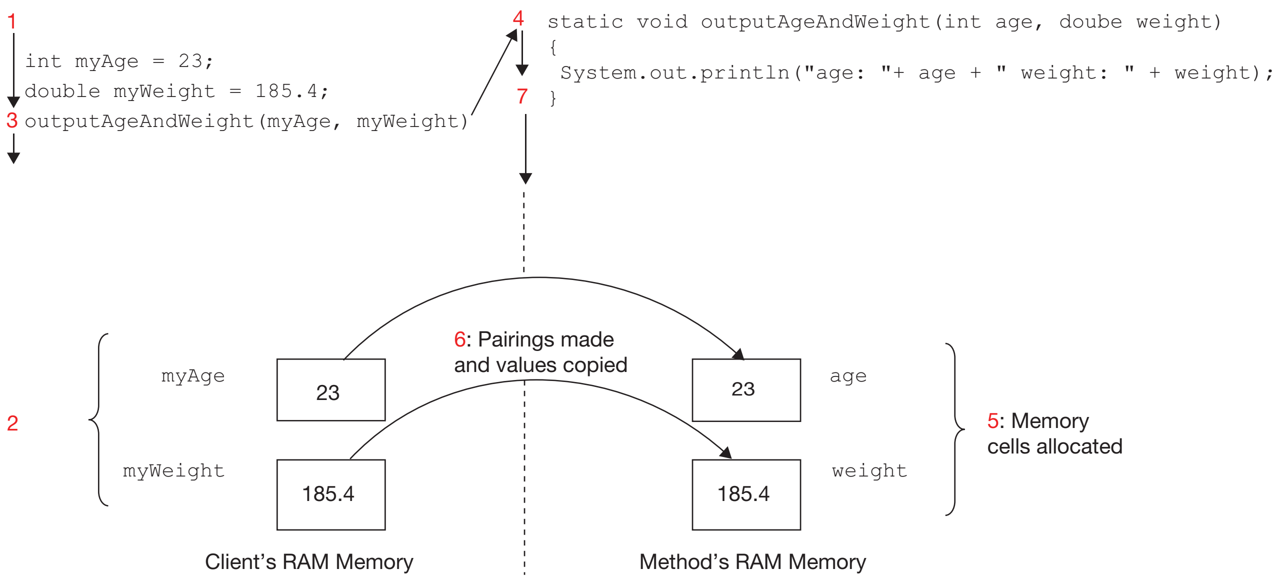


Figure 3.4
The transfer of information to a method via value parameters.

The dotted line in the figure is a line that the code in the client and worker methods cannot cross. The client code on the left of the figure cannot access the contents of the member cells `age` and `weight`, and the worker method code cannot access the memory cells `myAge` and `myWeight`. Inserting the statement

```
myAge = myAge + 1;
```

into the code of the method `outputAgeAndWeight` would result in a translation error because `myAge` is only known to the client code.

When the method completes its execution, the variables named in its parameter list are de-allocated, and their storage is returned to the memory manager. The result is that the values stored in these memory cells are lost, in that they are no longer available to the program. An understanding of this is fundamental to the notion of value parameters. It is also important to realize that the four memory cells created during events 2 and 5 are separate and distinct. To emphasize this, the names of the parameters (`age` and `weight`), coded in the worker method's signature, were intentionally chosen to be different than the names of the arguments passed to method (`myAge` and `myWeight`). Even if the argument and parameter names were the same (e.g., both coded as `age` and `weight`), event 5 would still create two distinct memory cells on the right side of Figure 3.4 named `age` and `weight`. A coding of the symbols `age` and `weight` in the worker method would refer the contents of these to memory cells, which would be de-allocated when the method ends its execution.

NOTE *Every time a method is invoked, the variables in its parameter list are allocated, and they are de-allocated when the method ends its execution.*

3.2.2 Scope and Side Effects of Value Parameters

The only way to pass information between arguments and parameters (i.e., between client code and worker method code) in Java is via value parameters, so it is important that we understand the limitations of the value parameter memory model presented in Figure 3.4 and its implications. As depicted in the figure, the client code has two variables it can access, `myAge` and `myWeight`, and the method has two variables it can access, `age` and `weight`. The client code cannot access the variables `age` and `weight`, and the worker method cannot access the variables `myAge` and `myWeight`.

In programming language jargon, we say that the worker method's variables `age` and `weight` are defined within the **scope** of its code, and the variables `myAge` and `myWeight` are out of its scope. A Java statement can only access variables that are within its scope.

Definition

The **scope** of a variable is the portion of a program in which it is defined and can therefore be accessed.

It is syntactically correct to make the argument names in a method invocation the same as the parameter names coded in the method's signature. For example, the names of the variables declared in the main method and the worker method's parameter list could both be named `myAge` and `myWeight`.

```
public static void main(String[] args)
{  int myAge = 23;
   double myWeight = 185.4;

   outputAgeAndWeight(myAge, myWeight);
}

static void outputAgeAndWeight(int myAge, double myWeight)
{
   System.out.println("age: "  + myAge + " weight: " + myWeight);
}
```

As previously mentioned, this coding of the method would still create two memory cells assigned to the method's parameters on the lower right side of Figure 3.4, but their names would now be `myAge` and `myWeight`. Now, the statement

```
myAge = myAge + 1;
```

written into the worker method's code would not result in a translation error because there is now a variable named `myAge` that is within its scope. The method's code would access the contents of its memory cell `myAge`, and the contents of that variable would be changed to 24. When the method endes and execution returnes to the client code, the contents of the client code's memory cell `my-Age` would be unchanged. It would still contain the value 23.

Normally, this is a good thing because it prevents an unwanted *side effect* of the method's code changing the client's data. Preventing this side effect assures that the values stored in the client code variables before the method was invoked will be the same values stored in those variables after the method completes it execution. In some cases, however, this is not what we want.

Consider the case of a method, named `swap`, that the client invokes to swap the contents of two of its variables, `a` and `b`, via the statements

```
int a = 10;
int b = 20;
swap(a, b);
System.out.println("a is: " + a + " and b: is " + b);
```

If the method is successful, the output produced after `swap` completes its execution should be `a` is 20 and `b` is 10.

The method `swap` would have two integer parameters to receive the values to be swapped, and its code would implement the swapping algorithm. The code of the method, preceded by a main method that invokes it, is given below:

```
public static void main(String[] args)
{
   int a = 10;
   int b = 20;

   swap(a, b);
   System.out.println("a is:" + a + " and b: " + b);
}

static void swap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Unfortunately, because Java uses value parameters, this coding of the method does not swap the client's variables, and the output produced by the last statement in the main method is `a` is 10 and `b` is 20.

Consistent with the memory model of value parameters, the client and the method each have their own memory cells named `a` and `b`. The method swaps the values contained in its two memory cells `a` and `b`, which has no side effects on the contents of the memory cells `a` and `b` allocated in the client code. These two cells remain unchanged (they still contain 10 and 20, respectively), which makes it appear that the swap algorithm was improperly coded in the worker method.

It turns out that it is impossible to write a method that swaps the contents of two client variables using primitive-type parameters because, by design, value parameters prevent a method from changing the values in the client's argument list. Some programming languages (e.g., C and C++) solve this problem by allowing another type of parameter called a reference parameter. Java does not support this type of parameter because it can lead to some undesirable side effects

3.2.3 Returned Values

A worker method can return one, and only one value to the method that invokes it. A method that returns a value is called a value returning or nonvoid method, and the key word **`void`** is not used in its signature. It is replaced with the type of the information the method returns. For example, the method `showInputDialog` in the API class `JOptionPane` is a nonvoid method that returns a reference to a `String`, so its signature contains the type `String` rather than the keyword **`void`**.

Methods can return the contents of reference variables, as the method `showInputDialog` does, or they can return the contents of primitive variables. In both cases, the returned value should be thought of as replacing the invocation of the method in the statement that invoked the method after the method executes. If the value is to be used later in the program, the invocation should be coded as the right part of an assignment statement that assigns the returned value to a variable.

For example, the two statements below prompt the user to enter a person's age and return the characters that are entered, but only the second one retains the location of the `String` object that is returned.

```
JOptionPane.showInputDialog("enter a person's age");
String sAge = JOptionPane.showInputDialog("enter a person's age");
```

A nonvoid method must contain a `return` statement or the method will not translate. The statement begins with the key word **`return`**, which is followed by the value that is to be returned. The value to be returned can be a literal, a variable, the value of an arithmetic expression, or a value returned from a method invocation. The following code segment contains a nonvoid method `multiply` preceded by the code of the main method that invokes it. The method `multiply` calculates and returns the product of two numbers passed to it.

```
public static void main(String[] args)
{
   double a = 10.0;
   double b = 20.0;

   double product = multiply(a, b);
   System.out.println(a + " x " +  b + " = " product);
}
```

```
static double multiply(double a, double b)
{
   double c;
   c = a * b;
   return c;
}
```

Because, as previously mentioned, an arithmetic expression can be coded in a `return` statement, the method could have been coded more succinctly as:

```
static double multiply(double a, double b)
{
   return a * b;
}
```

There can be more than one return statement in a nonvoid method, but we will not have a use for that feature of the language until we gain an understanding of the material presented in the next chapter.

3.2.4 Class Level Variables

Class level variables are another way of sharing information among methods. However, the manner in which the information is shared and the syntax used to code class level variables are both very different than when arguments parameters and `return` statements are used to share information.

To begin with, unlike using arguments and parameters to pass information to a worker method, in order for methods to share information using class variables, the methods must be coded in the same class. The information sharing is accomplished by coding the variable outside of the code blocks of the methods in the class. Good coding style dictates that they be coded at the top of the class before the code of any of the methods. When this done, the variable is within the scope of all of the methods in the class. The methods actually share the same variable, which permits any method in the class to fetch and overwrite the variable's contents. Unlike arguments and parameters, class variables provide a two-way path for methods to share information. One method can write a value into the variable, and another method can read the value from it.

Although we did not explain class variables in this much detail in Chapter 2, the program presented in Figure 2.12 used a class variable named `count` (line 8) to share the game's time between the method `timer` that was incrementing it (line 23), and the method `draw` that was outputting it to the game board (line18). As shown on line 8 when a class variable is used in the program's class, its declaration must begin with the key word `static`:

```
static int count = 0;
```

Aside from that, its declaration syntax is the same as that used to declare a variable inside of a method's code block.

A variable can be declared inside a method's code block that has the same name of a class-variable. When this is done, a memory cell is created with the same name as the class variable and is called a *local variable*. All uses of the variable's name inside the method's code block refer to the local variable, and the local variable can only be accessed by the method's code. To access the class variable from inside the method, the name of the variable would be preceded by the name of the class followed by a dot (just as when we invoke static methods).

Wherever possible, it is good programming practice not to use the names of class variables for naming variables declared inside of methods. For one thing, it reduces the program's readability because, if we fail to realize that the local variable is declared, we would erroneously believe that the class variable is being used inside the method. In addition, if we neglect to declare the local variable when coding the method, the translator will assume we want to use the class variable, and it will not remind us that we neglected to declare the local variable.

! TIP

It is good programming practice not to use the names of class variables for naming local variables declared inside of methods.

Figure 3.5 presents a program that contains four worker methods and demonstrates information sharing via value parameters, return statements, and class variables, the use of local variables, and the features of methods that make them reusable. The inputs to the program and the outputs the program produces are shown Figure 3.6.

Lines 30–36 contain the code of the method `inputInteger`. Its signature (line 30) indicates that it is a nonvoid method that returns an integer and has a `String` parameter named `prompt`. The method passes the string sent to it to an input dialog box (line 33) to be displayed as a `prompt` to the program user. The returned user input is parsed into the integer variable `a` (line 34), and then the parsed value is returned to the invoker (line 35). The inclusion of a string parameter in its signature allows the invoker to specify the prompt sent to the input dialog box. In addition, the method parses the input integer, freeing the invoker from that responsibility. Both of these features make this a highly reusable method. It is invoked five times within the program (on lines 11, 17, 18, 23, and 24), and each time it is sent a different prompt.

Two class variables, `a` and `b`, are declared on lines 4 and 5 with `a` initialized to 10. The value in the class variable `a` is included in the string passed to the invocation of `inputInteger` on line 11, which is displayed as the prompt in the input dialog box (Figure 3.6a) produced by line 33. The method `inputInteger` declares its own local variable `a` on line 32, so the assignment of the parsed value of the user input into the variable `a` (line 34) changes the contents of the local variable, leaving the class variable unchanged. This is verified by the first output (Figure 3.6b) produced by the program (lines 14–15), which indicates that a 10-year-old child will be 13 in 3 more years.

The method `dif` is invoked on line 13 to calculate the first output: the years to reach the desired age. It is passed the desired age, input on lines 11-12, as the first argument on line 13. Because the method main does not declare a local variable named `a`, the second argument on line 13, `a`, is the class variable. The method then calculates the difference between the two values passed to its two parameters (line 40), the desired age and the value stored in the class variable. Because the desired age was input as 13, and the output indicates that it will be reached in 3 more years, the class variable passed to the method's second parameter must have contained the value 10 at the time `dif` was invoked. The previous assignment into the local variable `a` on line 34 had no effect on it.

It should be noted that even if the parameter `a` was reassigned inside the method `dif`, the class variable would still retain the value 10 because Java uses value parameters. All references in the method `dif` to the variable `a` refer to the parameter, which can be thought of as a local variable.

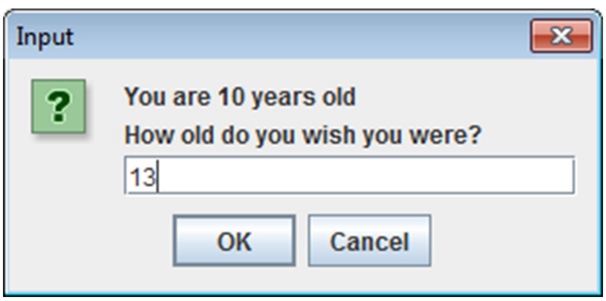
Two swap methods, `swapParameters` and `swapClassLevels`, are coded on lines 43–55. The first of these methods is invoked on line 19. It contains two parameters, `a` and `b` (line 43), to receive the values to be swapped, which are input on lines 17 and 18 (Figures 3.6 c and d). Although the names of the parameters are the same as the names of the class-level variables, they are not the same memory cells as the class variables. When the values stored in the parameters are swapped (lines 45–47), the output produced by lines 20–21 of the main method confirms a feature of value parameters: changes to parameters have no effect on the arguments sent to the method. The numbers output by the main method are output in the same order in which they were input: 1111 followed by 2222 (Figure 3.6 e).

Because the main method does not declare local variables named `a` and `b`, lines 23 and 24 store the values returned from `inputInteger` in the program's class variables (Figures 3.6 f and g). The second swap method, `swapClassLevels`, is invoked on line 25. It has an empty parameter list (line 50) and only one local variable, `temp`. Therefore, the variables `a` and `b` used in this method default to the class-level variables. Because these are shared with the main method, the swapping of the values in these variables performed on lines 52–55 does have an effect on the output produced by the main method (lines 26–27). As a result, the numbers input on lines 23 and 24 (8888 and 9999) are output in reverse order (9999 followed by 8888) by lines 26 and 27 (Figure 3.6 h).

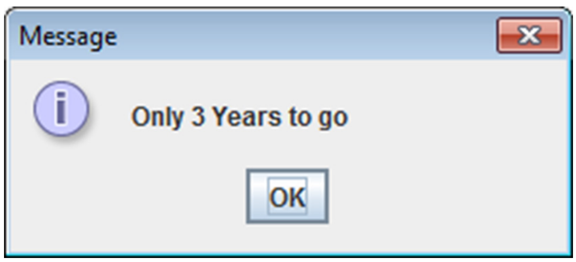
```
1   import javax.swing.JOptionPane;
2
3   public class MethodsAndParms
4   {   static int a = 10; // Two classlevel variables
5       static int b;
6
7       public static void main(String[] args)
8       {
9          int desiredAge, first, second, difference; // local Variables
10
11         desiredAge = inputInteger("You are " + a + " years old" +
12                                  "\nHow old do you wish you were?");
13         difference = dif(desiredAge, a);
14         JOptionPane.showMessageDialog(null, + "Only " + difference +
15                                              " Years to go");
16
17         first = inputInteger("Enter the first number to swap");
18         second = inputInteger("Enter the second number to swap");
19         swapParameters(first, second);
20         JOptionPane.showMessageDialog(null, "Swapped using parameters: " +
21                                             first + " " + second);
```

```

        a = inputInteger("Enter the first number to swap");
        b = inputInteger("Enter the second number to swap");
        swapClassLevels();
        JOptionPane.showMessageDialog(null, "Swapped using class " +
                                            "levels: " + a + " " + b);
    }

    static int inputInteger(String prompt)
    {
        int a; // a local variable
        String sInput = JOptionPane.showInputDialog(prompt);
        a = Integer.parseInt(sInput);
        return a;
    }

    static int dif(int desiredAge, int a)
    {
        return desiredAge - a;
    }

    static void swapParameters(int a, int b)
    {
        int temp = a;
        a = b;
        b = temp;
    }

    static void swapClassLevels()
    {
        int temp = a;
        a = b;
        b = temp;
    }
}
```

Figure 3.5
The application **MethodsAndParameters**.



(a)



(b)

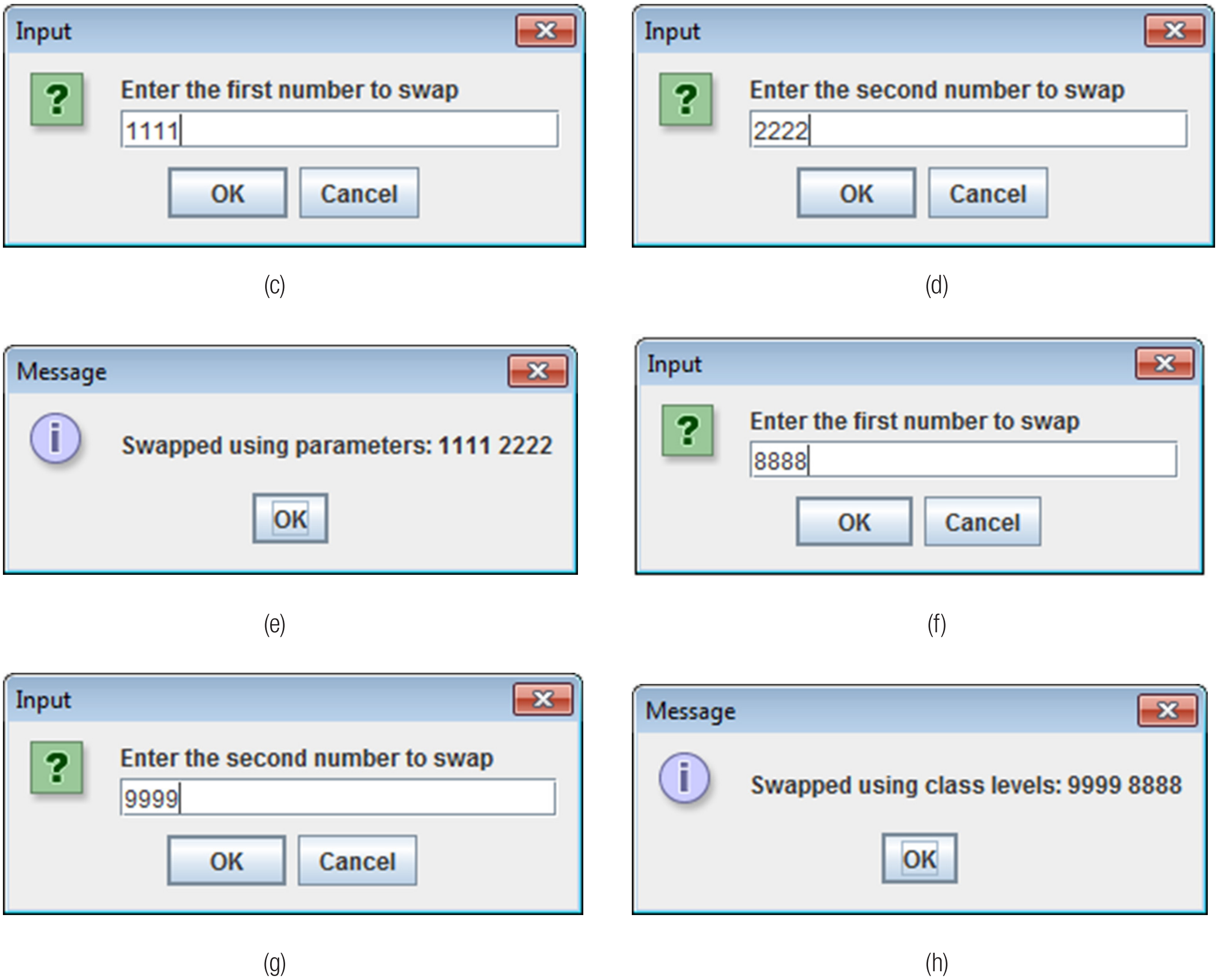


Figure 3.6
Inputs and resulting outputs produced by the application **`MethodsAndParameters`**.

3.3 THE API GRAPHICS CLASS

Having gained a deeper understanding of methods and the techniques for sharing information between methods and the program code that invokes them, we will reinforce those concepts in this section by examining several worker methods in the API `Graphics` class. As discussed in Chapter 2, this class contains methods for drawing text on `Graphics` objects. It also contains methods used to change the drawing color and for drawing lines, rectangles, ovals, and circles on `Graphics` objects.

3.3.1 Changing the Drawing Color

All drawing performed on a `Graphics` object is performed in the current color. The default current color is black. The `setColor` method in the `Graphics` class can be used to change the current drawing color. One argument, used to specify the new value of the current drawing color, is passed to the method. Table 3.1 gives the names of the thirteen predefined color variables in the class `Color`. Because these variables are static variables, they are referred to by their name preceded by `Color` followed by a dot. For example, to set the color of all subsequent drawings on the `Graphics` object `g` to red, we would code:

```
g.setColor(Color.RED);
```

As previously discussed, the `Graphics` object attached to our game board is passed into the `draw` method's parameter `g` when the game environment invokes the `draw` method. Therefore, if this statement were coded in the `draw` call back method, the current drawing color of the game board would be changed to red.

Table 3.1
Thirteen of the Predefined Colors in the Color Class

Color	Variable Name
black	BLACK
blue	BLUE
cyan	CYAN
dark gray	DARK_GRAY
gray	GRAY
green	GREEN
light gray	LIGHT_GRAY
magenta	MAGENTA
orange	ORANGE
pink	PINK
red	RED
white	WHITE
yellow	YELLOW

3.3.2 Drawing Lines, Rectangles, Ovals, and Circles

Figure 3.7 presents five of the methods in the `Graphics` class. These methods are nonstatic void methods. As their names imply, the first method is used to a draw line, and the remaining four methods are used to draw rectangles and ovals. To specify the location of the item to be drawn, all of the methods are passed (x, y) coordinates whose units are pixels.

The line drawing method, `drawLine`, is passed two sets of (x, y) coordinates, which are the endpoints of the line to be drawn. For example, to draw a line from (30, 50) to (60, 80) on the `Graphics` object `g`, we would code:

```
g.drawLine(30, 50, 60, 80);
```

The rectangle drawing methods `drawRect` and `fillRect` are used to draw the outline of a rectangle and to draw a filled (solid) rectangle, respectively. Their first two arguments specify the coordinates of the upper left corner of the rectangle, and the third and forth coordinates specify the width and height of the rectangle in pixels. For example, to draw the outline of a rectangle whose upper left corner is at (100, 200) and is 50 pixels wide and 75 pixel high on the `Graphics` object `g`, we would code:

```
g.drawRect(100, 200, 50, 75);
```

To draw this rectangle as a solid rectangle, filled with the current drawing color, we code:

```
g.fillRect(100, 200, 50, 75);
```

The method `drawOval` is used to draw the outline of an oval, and the method `fillOval` is used to draw a solid oval filled with the current color. These ovals are drawn within a specified rectangle (which is not drawn). The method's four parameters are identical to those of the rectangle methods previously discussed and are used to specify the rectangle's (x, y) location and its width and height. For example, to draw the outline of an oval 50 pixels wide and 70 pixels high inscribed inside a rectangle whose upper left corner is at (100, 200), we would code:

```
g.drawOval(100, 200, 50, 75);
```

To draw this oval as a solid oval filled with the current drawing color, we code:

```
g.fillOval(100, 200, 50, 75);
```

`drawLine(int x1, int y1, int x2, int y2)` Draws a line, using the current color, between the points `(x1, y1)` and `(x2, y2)`
`drawRect(int x, int y, int width, int height)` Draws the outline of a rectangle whose upper left corner is at (`x`, `y`) and whose width and height are `width and height`, using the current color
`drawOval(int x, int y, int width, int height)` Draws the outline of an oval bounded by the rectangle whose upper left corner is at (`x`, `y`) and whose width and height are `width` and `height`, using the current color
`fillRect(int x, int y, int width, int height)` Draws a rectangle whose upper left corner is at (`x`, `y`) and whose width and height are `width` and `height`, filled with the current color
`fillOval(int x, int y, int width, int height)` Draws an oval bounded by the rectangle whose upper left corner is at (`x`, `y`) and whose width and height are width and height, filled using the current color

Figure 3.7
Primitive-shape drawing methods in the **`Graphics`** class.

The oval drawing methods can be used to draw circles by making the third and fourth arguments, the height and width of the rectangle that encloses the oval, the same number of pixels. For example, to draw a solid circle 50 pixels in diameter inscribed inside a rectangle whose upper left corner is at (100, 200) we would code:

```
g.fillOval(100, 200, 50, 50);
```

When the statements presented in this section are coded in the game environment's `draw` call back method, the lines and shapes they draw appear on the game board because, as mentioned at the end of Section 3.3.1, the `Graphics` object passed into the `draw` call back method's parameter `g` is attached to our game board.

Figure 3.8 presents the application `LinesAndShapes` that draws two lines in the default color (black), two dark-gray rectangles, a red oval, and a blue circle on the game-board object. The graphical output of the program is shown in Figure 3.9. Lines 16–26 coded inside the draw method perform the drawing. Consistent with the variable names given in Table 3.1, the argument sent to the `setColor` method on line 19 has an underscore separating the words `DARK` and `GRAY`.

```
1     import edu.sjcny.gpv1.*;
2     import java.awt.*;
3
4     public class LinesAndShapes extends DrawableAdapter
5     {
6       static LinesAndShapes ga = new LinesAndShapes( );
7       static GameBoard gb = new GameBoard(ga, "Lines and Shapes");
8
9       public static void main(String[] args)
10      {
11        showGameBoard(gb);
12      }
13
14      public void draw(Graphics g) // the drawing call back method
15      {
16        g.drawLine(100, 75, 260, 75);     //Lines
17        g.drawLine(300, 50, 400, 100);
18
19        g.setColor(Color.DARK_GRAY);
20        g.drawRect(100, 170, 100, 60);    //Rectangles
21        g.fillRect(280, 170, 150, 40);
22
23        g.setColor(Color.RED);
24        g.drawOval(55, 300, 180, 80);     //Ovals
25        g.setColor(Color.BLUE);
26        g.fillOval(280, 300, 100, 100);
27
28      }
29    }
```

Figure 3.8
The application **`LinesAndShapes`**.

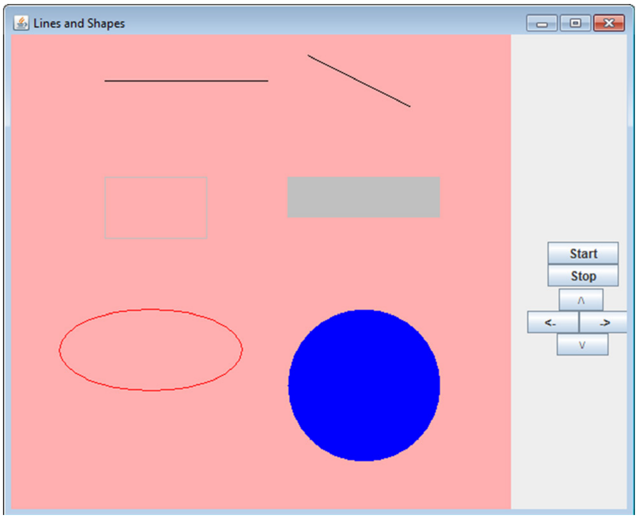


Figure 3.9
The output produced by the application **`LinesAndShapes`**.

3.4 OBJECT ORIENTED PROGRAMMING

Early programming languages were designed in the procedural paradigm. In this paradigm, a program is decomposed into smaller parts called subprograms, and the language provides a mechanism for combining the subprograms into the larger program. During the design process, the programmer focuses on the definition of the subprograms and how the program's data will be stored. In this paradigm, the subprograms and the program's data are separated and coded into two distinct entities.

Object oriented programming is a more recent programming paradigm. The paradigm is an attempt to facilitate the development of programs that deal with objects, such as starships, or people, or Web pages. In this paradigm, the program is decomposed into the various classes to which the objects belong. During the design process, the programmer focuses on determining the objects the program will deal with, the attributes of each object (e.g., a starship's name), and the operations that can be performed on each object (e.g., changing a starship's location). In this paradigm, the operations (subprograms) and attributes (data) are collected and coded into one entity called a class.

Both paradigms are in use today. Some programming solutions are better designed and more easily implemented using the procedural paradigm, and others are more easily designed and implemented using the object paradigm. C is a procedural language, C++ is a language that can be used in both the procedural and object paradigm, and Java is an object oriented language. If the program deals with objects, then the object paradigm should be strongly considered.

3.4.1 What Are Classes and Objects?

A **class** is a blueprint of how to construct an item, and an **object** is a particular item or instance of a class. For example, we all belong to the class human. That class contains a genetic blueprint of how to construct a human object, which we call a person. As per the human blueprint, all people have common attributes. For example, all people have colored-eyes, colored-hair, and eventually grow to an adult height. But clearly, all people (except identical twins) are also different. For example, people grow to different heights, have different-colored eyes, and different hair colors, what makes objects different is that each object contains its own value of the attributes contained in the blueprint.

Definition

A **class** is a blueprint of how to construct an item, and an **object** is a particular item or instance of a class.

The value of Mary's three attributes could be 63 inches for height, blue for eye color, and red for hair color. The value of her sister Kate's attributes could be 68 inches for height, brown for eye color, and black for hair color. No wonder these two objects look different. Now suppose Kate, who always admired her sister's red hair and blue eyes, dyed her hair to a red color and inserted a set of blue contact lenses into her eyes. These hair coloring and lens insertion operations would change the values of two of Kate's attributes, and she would then look like a taller version of Mary.

In object oriented languages, a class is the mechanism for defining the blueprint of an entity. As such, it contains the attributes that each object in the class will have (e.g., height, hair color, and eye color). To store the different values of these attributes for each object constructed from the blueprint, the attributes are represented within the class as variables. In addition, because the values of the attributes of an object can change, the class contains methods (e.g., `setEyeColor` and `setHairColor`) that can operate on the variables to change, or set, the values they store to new values.

3.5 DEFINING CLASSES AND CREATING OBJECTS

During the design process of an object oriented program, the programmer focuses on determining the objects the program will deal with, the attributes of each object, and the operations that can be performed on them. The blueprint for each type of object will be coded into a programming construct called a class that will represent the attributes as variables and the operations as methods. In the remainder of Section 3.5, we will discuss a graphical tool used to specify a class and the Java syntax used to code that specification into a Java program.

NOTE *The programming construct* **class** *is comprised of variable definitions and method definitions.*

3.5.1 Specifying a class: Unified Modeling Language Diagrams

A unified modeling language (UML) diagram is a graphical representation of a class. The diagram consists of three rectangles stacked on top of each other. From top to bottom, as shown in Figure 3.10, these rectangles are used to specify the class name, the variables that will be part of the class, and the class's methods. The variables are called data members (of the class), and the methods are called member methods because they are both part of (members of) the class being specified.

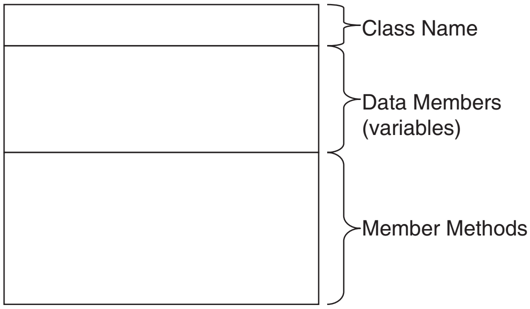


Figure 3.10
The template of a UML diagram.

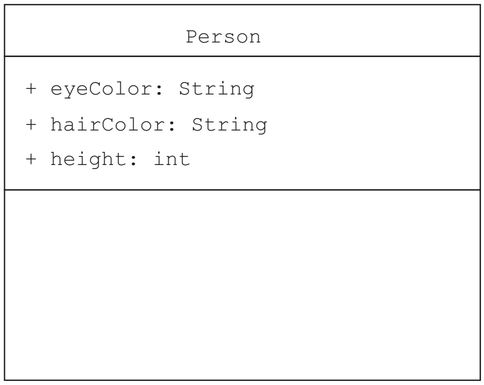


Figure 3.11
The specification of the class Person, Version 1.

As an example, consider a program that is going deal with people objects where each person has three attributes: eye color, hair color, and height. The UML diagram used to specify the class, whose name was chosen to be `Person`, is shown in Figure 3.11. The name of the class appears at the top of the diagram, and the class's three data members are tabulated in the second box of the diagram.

To succinctly convey information about a class's data members and methods, UML diagrams employ a standardized notation, some of which is included in Figure 3.11. For example, the type of each data member is specified by following its variable name with a colon and the type of the variable. As shown in the figure, the class `Person` has two String data members and one integer data member.

Non-primitive data member variables, such as `eyeColor` and `hairColor` in the class `Person`, are referred to as *instance variables*, because they will store the address of an instance of a class. More specifically, `eyeColor` will store the address of an instance of the class String. Just as we say that the variable height is an integer, we say the variable `eyeColor` is an *instance* of a `String`.

The three data members of the class `Person` are preceded with a plus (+) sign. The plus sign is used to denote the *access* property of a data member of a class. When the UML specification of a class is coded into a Java class construct, the + sign is coded as the key word **`public`**. Another alternative is to precede the names of the data members with a minus (–) sign, which is coded as the key word **`private`**. We will learn more about the implications of the use access modifiers in the next section.

3.5.2 The Class Code Template

Like the data members of a class, a class itself can be public or private, although most classes are public. We will discuss private classes in Chapter 7. The code template for a public Java class is given below:

```
public class ClassName
{
   //data members are coded here
   //member methods are coded here
}
```

The name of the class, given at the top of the UML diagram, is substituted for `ClassName` on the first line of the template. The declaration of the class's data members and the code of its member methods are coded inside of a pair of braces that make up the class's code block. The variables that represent a class's data members are coded before the code of the class's member methods. While this is not a Java syntax rule, it is considered good coding practice. The code of the class specified by the UML diagram presented in Figure 3.11 is given in Figure 3.12 with the data members set to initial values.

```
public class Person
{
   //data members
   public String eyeColor = "blue";
   public String hairColor = "red";
   public int height = 65;

   //member methods

}
```

Figure 3.12
The code of the class Person specified in Figure 3.11.

Normally, the initial values are chosen to be the most common value of the data members. In this case, the assumption is that most people have blue eyes, red hair, and are 65 inches tall.

3.5.3 Creating Objects

In Section 2.5, we examined a two-line syntax for declaring objects in the class `String`. For example:

```
String firstName;
firstName = new String("John");
```

The first line creates the reference variable `firstName` that can store the address of a `String` object, which is initialized to the default value `null`. The second line creates a new `String` object, stores the string `John` "`John`" inside of it, and then overwrites the `null` value stored in the variable `firstName` with the address of the object. Alternately, the two lines of code can be consolidated into one line:

```
String firstName = new String("John");
```

When talking about the object created with either the two- or one-line syntax, in the interest of brevity we say that "we created a string object named `firstName`," or we might be asked to "output the object `firstName`." However, experienced programmers know that it is more accurate to say, "we created a string object referenced by the variable `firstName`," or "output the object referenced by `firstName`." With that understanding, we will use the brief version in the remainder of this text.

By changing the name of the class coded in the one- or two-line syntax and usually the code inside the parentheses, both versions of the syntax can be used to declare an object in any class. The code fragment below uses the one-line grammar to create two `Person` objects, one named `mary` and the other named `kate`:

```
Person mary = new Person();
Person kate = new Person();
```

The memory allocated by these two statements is shown in Figure 3.13. As defined in the `Person` class (Figure 3.12), each object contains the same three variables set to initial values. The variables `hairColor` and `eyeColor` store the address of string objects that contain the initial values. At this point, Mary and Kate are identical twins and will remain so until we add methods to the class that can change the values of the data members.

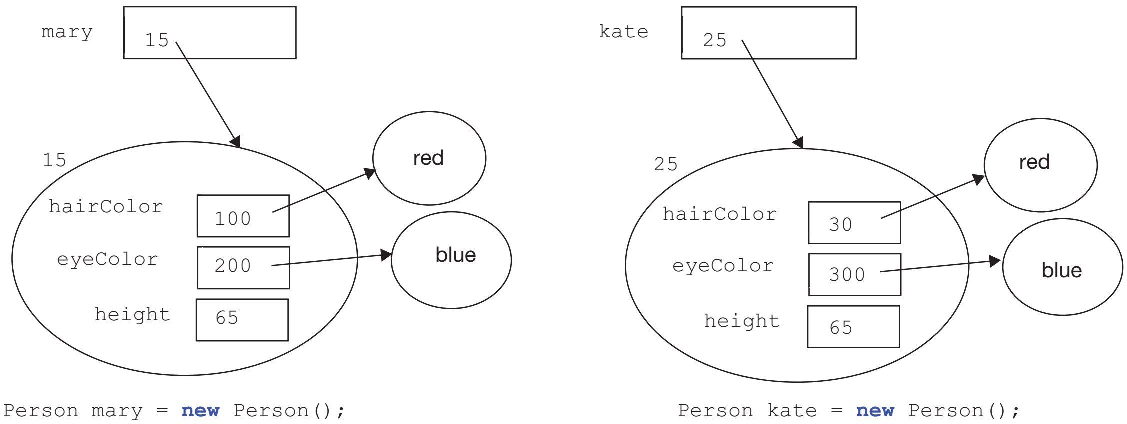


Figure 3.13
Two Person objects and the statements that constructed them.

Constructor methods

Let us assume that the two lines of code that created the objects `mary` and `kate`

```
Person mary = new Person();
Person kate = new Person();
```

were coded in the main method of a program that dealt with people, or more precisely, `Person` objects. These two lines of code should be thought of as the main method's request to the class to create two `Person` objects. If I call a carpenter and request that he create, or construct, a shed for me, I become his client. The two terms, *construct* and *client*, used in this analogy are used in the programming jargon of classes and objects. We would say that the class `Person` has constructed two objects for the client code main. Any section of code that declares an object in a class is considered to be client code (of that class), and the class is said to have constructed the objects for the client code.

Every class has at least one member method that constructs new objects. These non-void methods are called *constructors*, and they execute every time the object declaration syntax is used to declare an object. During the execution of a constructor method, the storage is allocated for the class's data members, the initial values are stored in the data members, the collection of data members is assigned a memory location (considered to be the location of the object), and that location is returned to the client code. The assignment operator included in the object-declaration grammar then stores the returned location of the object in the object's reference variable (e.g., `mary`).

The name of a constructor method is always the name of its class, so the code to the right of the key word **new** in the declaration of `mary`'s object

```
Person mary = new Person();
```

is actually an invocation to the constructor method named `Person` that has no parameters and returns the address of a newly created `Person` object.

The class `Person` shown in Figure 3.12 does not contain a constructor method, which in this case would be a method named `Person`. When a class does not contain a constructor, a Java-provided constructor method is used to construct objects. This constructor, referred to as the default constructor, has no parameters, and it performs the functions previously mentioned:

1. Allocates the storage for the data members of the class
2. Stores the initial values in the data members
3. Assigns the collection of data members a memory location
4. Returns the location to the client code

Because the class `Person` does not contain any methods, the two `Person` objects `mary` and `kate` declared as

```
Person mary = new Person();
Person kate = new Person();
```

would be created by the default constructor, which would perform the four functions listed above. Each object's data members would be set to the initial values specified in the data-member portion of their class (Figure 3.12) during function 2. In Section 3.6.2, we will learn how to add constructor methods that we write to a class. These methods can contain parameters and code to extend the four functions performed by the default constructor.

3.5.4 Displaying an Object

Objects can be displayed to the system console and to a graphical game board in one of two ways. We can simply mention the name of the object or invoke the `toString` method on the object inside a method invocation used to display strings, or we can add a method to the object's class that, when invoked, outputs the object. The following statements use the first approach to display the `Person` object `mary` to the system console object, `System.out` and then to a `GameBoard` object named `g` at location (210, 100).

```
System.out.println(mary);
g.drawString(mary.toString(), 210, 100);
```

This approach is usually not too interesting because what is displayed is the location of the object that is stored in the reference variable `mary` preceded by an ampersand (@) and the name of the object's class. Referring to Figure 3.13, Mary's object is located at address 15, so the output to the system console and the game board would be `Person@F`. The second alternative, adding a method to the object's class that when invoked outputs the object, produces a more interesting and useful output. This technique will be discussed in Section 3.6.

Figure 3.14 presents an application that creates two `Person` objects, whose class is defined in Figure 3.12, and outputs their locations to the system console and the game board. The output is shown in Figure 3.15, except that the program does not produce the more interesting output of the actual object `Mary` shown in the middle of the game board. As previously mentioned, the techniques for producing that output will be discussed Section 3.6, and the code to produce the output will be left as an exercise for the student.

Lines 7, 10, and 11 create two `Person` objects using the two-line syntax. The reference variables `mary` and `kate` are declared as class variables on line 7, so they can be accessed from the main method and the `draw` call back method. Lines 13–14 and lines 21–22 output the object's locations to the system console and the game board, respectively. These locations can be thought of as the locations assigned to the two objects by Java's memory manager when lines 10–11 execute.

```
1   import edu.sjcny.gpv1.*;
2   import java.awt.*;
3   public class ClassAndObjectBasics extends DrawableAdapter
4   {
5      static ClassAndObjectBasics ge = new ClassAndObjectBasics();
6      static GameBoard gb = new GameBoard(ge, "Class & Object Basics");
7      static Person mary, kate;
8
9      public static void main(String[] args)
10     {  mary = new Person();
11        kate = new Person();
12
13        System.out.println(mary);
14        System.out.println(kate);
15
16        showGameBoard(gb);
17     }
18
19     public void draw(Graphics g)
20     {
21        g.drawString(mary.toString(), 210, 100);
22        g.drawString(kate.toString(), 210, 120);
23     }
24  }
```

Figure 3.14
The application **`ClassAndObjectBasics`**.

System console output:
Person@3a6727
Person@4a65e0

Game board output:

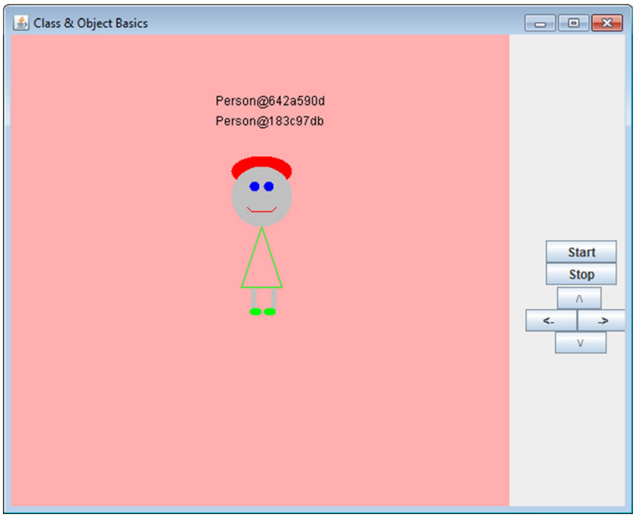


Figure 3.15
The object locations output by the application **ClassAndObjectBasics.**

3.5.5 Designing a Graphical Object

During the specification and design of a program we identify the types of objects that the program will deal with and the operations to be performed on them. Then during the program development process, a class is developed for each of the object types, and the operations performed on the objects become the methods of the class. A common operation performed on objects is to display them, and in the previous section we were able to display a `Person` object's location without adding a method to the `Person` class. To produce the more interesting display of an object, such as the drawing of the object `mary` with her red hair and blue eyes shown in the middle of Figure 3.15, we add a method to the object's class that uses the shape and line drawing methods of the `Graphics` class to produce the output.

In Section 3.6, we will learn how to add this method, whose name usually begins with the prefix `show`, and other methods that perform common operations on objects to an object's class. In preparation for the coding of this method, the programmer has to design each type of graphical object using the basic drawing shapes available in the `Graphics` class. In this section, we will become familiar with techniques used to design these objects so that the drawing can be easily coded into a `show` method, and the object can be easily manipulated by other class methods that perhaps relocate the object, erase the object, or animate the object.

Drawing an Object

To begin, we draw a picture of the object using the graphical shapes available in the API `Graphics` class discussed in Section 3.3. The object should be inscribed in a rectangle, and if ovals are used, they should be inscribed in their own rectangle. Figure 3.16 shows a sketch of a snowman-type object comprised of two rectangles and two circles. The rectangles that inscribe the ovals and the entire object are shown in red.

The dimensions, in pixels, of each of the basic shapes that make up the game piece should be given in the drawing. For example, the snowman's hat is specified on the upper right side of the figure to be 10 pixels wide and 15 pixels high. Ovals that are circles, such as a snowman's head and body, can be specified with one dimension (e.g., the diameter of the snowman's head is 20 pixels). After the dimensions of all of the shapes that make up the object have been noted on the drawing, the overall width and height of the inscribing rectangle is added to the drawing. This is shown in Figure 3.16 on the bottom and left side of the inscribing rectangle. The width is simply the width of the snowman's body, 40, and the height is the sum of the heights of the shapes that make up the snowman, 77 (15 + 2 + 20 + 40).

The color of each shape that makes up the object should also be given, as shown in the lower right portion of the figure. The word default used in the object's color specification implies that snowmen objects could be constructed with hats that are not black. Finally, a point that will be used to locate the object on the game board is noted on the drawing. As shown in the upper left side of the figure, this point is typically the upper left corner of the inscribing rectangle.

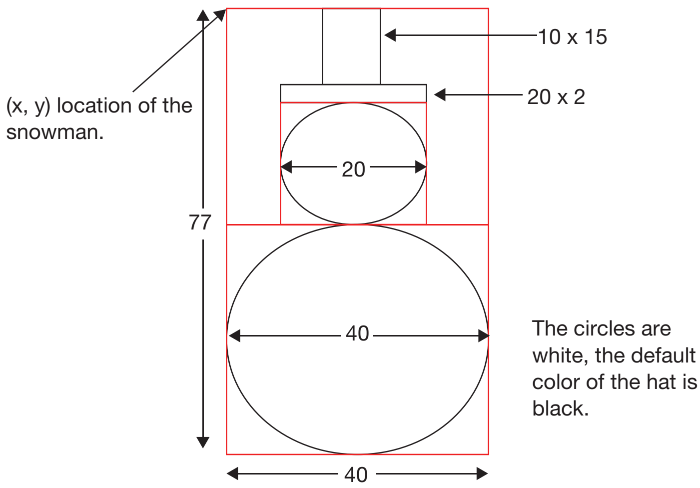


Figure 3.16
The design of a snowman game piece.

The next step in this design process is to determine the locations of each of the shapes' upper-left corner relative to the (x, y) location of the game piece, which for our snowman is the upper left corner of the inscribing rectangle. These locations, along with the width and height of each of the shapes, are entered into in a table that will be used in the `show` method to draw each of the shapes. The table is a digital representation of the object, and the process of determining the data is referred to as digitizing the object.

The digital representation of our snowman object is given in Table 3.2. As previously mentioned, each location in the table is relative to the (x, y) location of the upper corner of the rectangle that inscribes the snowman object. Therefore, each location given in the table begins with an x or y followed by the x or y distance to the corner of the shape. When reading the locations in the table, it should be remembered that the positive x direction is to the right, and the positive y direction is down.

To determine these distances, we either consider the dimensions of each of the shapes given in Figure 3.16, or we draw the object on a piece of graph paper whose origin is the upper left corner of the object's inscribing rectangle. Then, the x and y distances to the upper-left corner of each shape

Table 3.2
Digital Representation of the Snowman Object Shown in Figure 3.16

Component	Shape	Shape's X or Line's X_1 Coordinate	Shape's Y or Line's Y_1 Coordinate	Width or Line's X_2 Coordinate	Height or Line's Y_2 Coordinate
Hat	Rectangle	x + (20 – 5)	y	10	15
Hat Brim	Rectangle	x + (20 – 10)	y + (15)	20	2
Head	Circle	x + (20 – 10)	y + (15 + 2)	20	20
Body	Circle	x	y + (15 + 2 + 20)	40	40

are simply the x and y coordinates of the shape's inscribing rectangle's upper left corner. The following two examples illustrate the technique of determining the x and y coordinates of each shape by considering the dimensions given in Figure 3.16.

1. To determine the x location of the upper left corner of the snowman's hat, which is x + (20-5) as indicated in the first row and third column of the table, half the width of the snowman's inscribing rectangle (20) is added to x because the center of the hat is at the center of the inscribing rectangle. Half the width of the hat (5) is subtracted from x because the left side of the hat is half the width of the hat closer to the left side of the snowman's inscribing rectangle than is the center of the hat.
2. To determine the y location of the upper left corner of snowman's head, which is y + (15+2) as indicated in the third row and fourth column of the table, the height of the hat (15) and the height of the hat brim (2) are added to y.

It should be noted that when lines are used in the object's drawing, each line is entered on a separate row of the table and, as indicated in the column headings of Table 3.2, the coordinates of the endpoints of the lines are entered into the rightmost four columns of the table. During the design phase of the program, a table is produced for each type of object in the program.

3.6 ADDING METHODS TO CLASSES

Many of the methods added to a class perform operations on the objects in the class. Some of these operations are so commonly performed, such as a method to display an object or a method to change the value of a data member, that they are included in most classes. To improve program readability, the names of these methods usually begin with a designated prefix. For example, the names of methods that display objects usually begin with the prefix `show`, and a method that begins with the prefix `set` usually changes the value of one of an object's data members.

In this section, we will study the techniques for adding methods to classes and how to use them to perform operations on objects. These methods will be added to a class named `SnowmanV1` that defines the object depicted in Figure 3.16 and digitized in Table 3.2. Its UML diagram will be progressively developed, by adding methods to it, as we move through the next few sections of this chapter.

Our starting point will be the UML diagram shown in Figure 3.17, which is implemented in Figure 3.18. The class has three data members, two integers, and a reference variable used to specify the location of a `SnowmanV1` object and its hat color. As shown in the UML diagram, the data member `hatColor` will refer to a color constant in the class `Color`. The code of the class `SnowmanV1` is presented in Figure 3.18. The three data members are coded on lines 4–6. The default location of the snowman is (5, 30), which is the upper left corner of the game board. The hat color has been initialized to the color constant BLACK.

SnowmanV1
+ x: int + y: int + hatColor: Color

Figure 3.17
The UML diagram of the class **SnowmanV1**

```
1   public class SnowmanV1
2   {
3      //data members
4      public int x = 5;
5      public int y = 30;
6      public Color hatColor = Color.BLACK;
7   }
```

Figure 3.18
The Class **SnowmanV1**

The use of a graphical snowman in this chapter will make the concept of operating on an object less abstract, and therefore more easily understood. For example, rather than the class' `show` method simply displaying an object's (x, y) location, its `show` method will display the object in a more tangible way: by drawing it on the game board at its (x, y) location. Rather than simply outputting the increased value of an object's x location, we will see the object move to the right.

3.6.1 The show Method

A method that begins with the prefix `show` is used to display an object and is a nonstatic method. Because the drawing methods of the `Graphics` class cannot be used to draw on the system console, we have to define what it means to show an object on the system console. The commonly accepted meaning is that the output would consist of the annotated values of the object's data members. This version of a show method, named `showXYToSC` (SC for system console) would invoke the `println` method and pass it a string that concatenates the annotation and the class's x and y data members. For example:

```
public void showXYToSC()
{
   System.out.println("x is: " + x +
                      "\ny is: " + y);
}
```

After this code is added to the class then snowman `sm1` could be output to the system console using the statement:

```
sm1.showXYToSC();
```

which would produce the output:

x is: 5

y is: 30

The statement `sm1.showXYToSC();` can be read in three different ways:

1. The object `sm1` is invoking the `showXYToSC` method.
2. The method `showXYToSC` is invoked on the object `sm1`.
3. The `showXYToSC` method is operating on the object `sm1`.

All three of these are synonymous; that is, they mean the same thing. In general, to cause a method to operate on an object, we precede the name of the method with the name of the object followed by a dot.

An important point to remember is that if we just focused on the code of the `showXYToSC` method and asked the question, when it mentions the data members `x` and `y`, which object's data members is it talking about, the answer is the data members of the object that invoked it. The true meaning of the statement "a method operates on an object" is that all occurrences of the names of the data members coded inside the method refer to the data members of the object that invoked it.

Figure 3.19 presents the expanded UML diagram that reflects the addition of two show methods, `showXYToSC` and `show`. The method `show` will use the digitized version of a snowman, presented in Table 3.2, to display a `SnowmanV2` object on the game board. Because the `shape` drawing methods in the `Graphics` class will need access to the game board object, the `show` method will have one parameter: a reference to a `Graphics` object. The characters `g: Graphics` that appear inside the parentheses of this method in the UML diagram is UML notation to indicate that this method has one parameter named `g` that is a reference to a `Graphics` object.

The code of this expanded class is given in Figure 3.20. A client application that declares a `SnowmanV2` object and outputs the object's address to the system console and the object to both the system console and the game board is shown in Figure 3.21. The application's output is shown in Figure 3.22.

SnowmanV2
+ x : int + y : int + hatColor: Color
+ showXYToSC() + show(g: Graphics)

Figure 3.19
The UML diagram of the class **`SnowmanV2`**.

The client code (Figure 3.21) invokes the default constructor on line 7 to declare a `SnowmanV2` object named `sm1`. The declaration uses the one-line object declaration syntax and is at the class level, so the object can be accessed by the `main` method and the `draw` call back method. Line 11 outputs the object's location to the system console, and line 12 invokes the `showXYToSM` method of the `Snowman`'s class to display snowman `sm1` to the system console. This method is coded on lines 12–16 of the `Snowman` class (Figure 3.20).

Line 19 of the application invokes the `SnowmanV2` class's `show` method to display snowman `sm1` on the game board. This invocation is coded in the `draw` call back method for two reasons. First, the `show` method must be passed a `Graphics` object on which to perform its drawing, and the `draw` method is the only call back method that is passed a `Graphics` object when it is invoked. It passes the Graphics object to the `show` method as an argument on line 19. Secondly, when the game board needs to be redrawn, the game environment invokes `draw`, which will then invokes `show` to redraw the snowman.

Lines 18–26 of Figure 3.20 are the code of the `show` method. The method's signature (line 18) includes the parameter `g` specified in the class's UML diagram (Figure 3.19). It uses this parameter to invoke methods in the `Graphics` class used to change the current drawing color (`setColor` lines 20 and 23) and to draw the rectangles and circles specified in Table 3.2. All of the shape locations sent to the `Graphics` class methods as arguments on lines 21–25 are those contained in the table. They contain the variables x and y because they are relative to the upper left corner of the rectangle that inscribes the snowman. Because the method does not declare local variables named `x` and `y`, the class-level data members `x` and `y` are used in these arguments. As a result, the snowman is drawn as shown in Figure 3.21 with the upper left corner of its inscribing rectangle at (5, 30).

```
1    import java.awt.Color;
2    import java.awt.Graphics; //needed for drawing shapes
3
4    public class SnowmanV2
5    {
6      //data members
7      public int x = 5;
8      public int y = 30;
9      public Color hatColor = Color.BLACK;
10
11     //member methods
12     public void showXYToSC()
13     {
14       System.out.println("x is: " + x +
15                          "\ny is: " + y);
16     }
17
18     public void show(Graphics g) //g is passed to the method
19     {
20       g.setColor(hatColor);
21       g.fillRect(x + 15, y, 10, 15);  //hat
22       g.fillRect(x + 10, y + 15, 20, 2); //brim
23       g.setColor(Color.WHITE);
24       g.fillOval(x + 10, y + 17, 20, 20); //head
25       g.fillOval(x, y + 37, 40, 40);    //body
26     }
27   }
```

Figure 3.20
The class **SnowmanV2**.

```
import edu.sjcny.gpv1.*;
import java.awt.Graphics;
public class ShowMethods extends DrawableAdapter
{
  static ShowMethods ga = new ShowMethods( );
  static GameBoard gb = new GameBoard(ga, "Show Methods");
  static SnowmanV2 sm1 = new SnowmanV2();

  public static void main(String[] args)
  {
    System.out.println(sm1);
    sm1.showXYToSC();

    showGameBoard(gb);
  }

  public void draw(Graphics g) //the drawing call back method
  {
    sm1.show(g);

  }
}
```

Figure 3.21
The application **ShowMethods**.

System Console Output
SnowmanV2@3a6727
x is: 5
y is: 30
Graphical Output

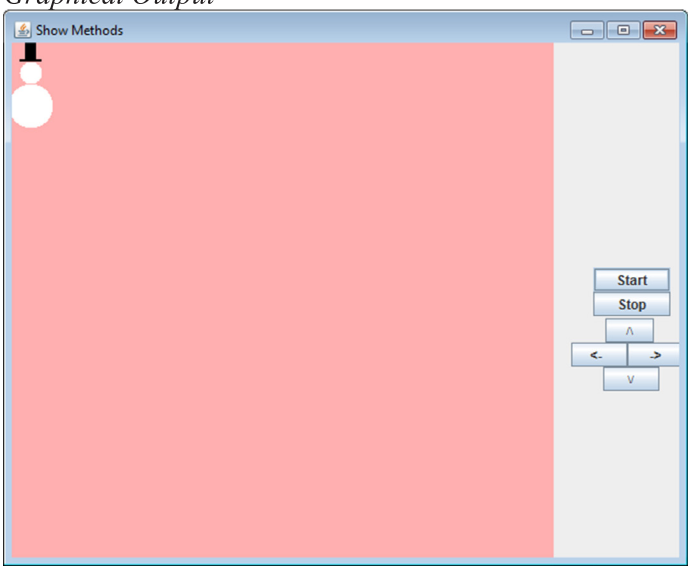


Figure 3.22
The console and graphical output produced by the application **ShowMethods**.

3.6.2 Constructors and the Keyword `this`

In Section 3.5.3, we learned that constructors are methods that construct objects, and the names of these methods must be the same as the class of the objects they construct. If a constructor method is not included in the specification and code of a class, a Java provided default constructor creates the object by performing the following four functions:

1. Allocate the storage for the data members of the class
2. Set the initial values into the data members
3. Assign the collection of data members a memory location
4. Return the location to the client code

Because the class `SnowmanV2` does not contain a constructor method, the default constructor is used to create all instances of this class (objects declared in this class). As a result, function 2 would locate them all at (5, 30), and they would all have black hats. When displayed, they would be displayed on top of each other giving the appearance that only one of them was displayed.

To allow the client code to specify the values of the data members of a newly constructed object, we add a constructor method to its class. Its code template is the same as the template used to code any other method, except its name must be the same as the class's name, and its signature cannot contain a return type. Its signature can contain a parameter list, and it can contain Java statements in its code block. When a constructor method is included in the code of a class, the default constructor is no longer available to construct objects in the class.

Figure 3.23 presents the UML diagram of the class `SnowmanV3` that contains a two-parameter constructor, which is a constructor with a parameter list that contains two parameters. When the client code uses this constructor method to create an object, just before the fourth function normally performed by the default constructor is performed, the constructor method executes. Storage is allocated for the constructor's parameters, the values of the client's arguments are copied into them, and then the code of the constructor executes.

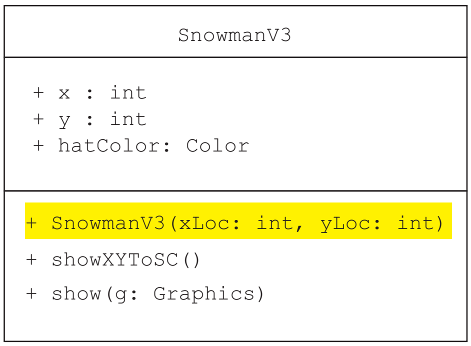


Figure 3.23
The UML diagram of the class **`SnowmanV3`**.

As with any method, the values copied into the parameters of the constructor could be used anywhere in the constructor's code block by coding the names of the parameters. It is often the case that these parameters are used by the client code to specify the initial values of the data members. When this is the case, the constructor's code block simply assigns the parameters to the data members:

```
public SnowmanV3(int xLoc, int yLoc)
{   x = xloc;
    y = yLoc;
}
```

After this two-parameter constructor's code is included in the code of the class `SnowmanV3`, the client could use the following code to declare two snowmen located at the upper right and lower left corners of the game board:

```
SnowmanV3 sm1 = new SnowmanV3(5, 30);
SnowmanV3 sm1 = new SnowmanV3(460, 423);
```

The Keyword `this`

A method in a class can contain a parameter whose name is the same as the name of one of the class's data members. When this occurs, we say that the parameters *shadow* the data members. For example, the signature of the `SnowmanV3` class's two-parameter constructor could have been coded as:

```
public SnowmanV3(int x, int y)
```

When the parameter names shadow data member names, the use of the name within the constructor refers to the parameter, not to the data member. As previously stated, parameters should be considered to be local variables. An assignment into the variable `x` within the constructor's code body changes the value stored in the parameter, not the value stored in the class-level data member `x`. We can refer to the data member within the code of constructor (or any other member method whose parameter list employs shadowing), by preceding the data member's name with the key word **`this`** followed by a dot, e.g., **`this`**`.x`. This syntax could be thought of as the variable `x` that is a data member of *this* class. Using this syntax, the `SnownanV3` class's two-parameter constructor could be coded as:

```
public SnowmanV3(int x, int y)
{   this.x = x;
    this.y = y;
}
```

This coding of the two-parameter constructor, which uses shadowing, is actually preferred when the parameter list is being used to reset the initial values of the data members. When shadowing is used, the name and type of the parameters and the class's data members in the UML diagram (Figure 3.24) are the same, which is a cue to anyone looking at the UML diagram that the constructor will reset the initial values of the data members.

Figure 3.25 is the implementation of the `SnowmanV3` class specified in Figure 3.24. A client application that declares and displays two instances of the class is shown in Figure 3.26, and the output to the game board produced by the application is shown in Figure 3.27.

The two-parameter constructor is coded on lines 12–16 of Figure 3.25. It is good programming style to code the constructor as the first method after the data member declarations. Because the names of the constructor's parameters are the same as the class's data members, the keyword **`this`** is used on lines 14 and 15 to access the class's data members and assign the initial values passed into the parameters to them.

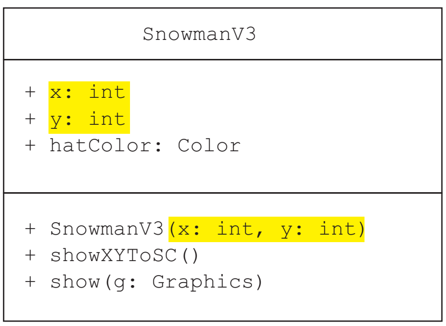


Figure 3.24
The modified UML diagram of the class **`SnowmanV3`**.

The client code (Figure 3.26) invokes the constructor twice, once on line 9 and again on line 10, to create two snowmen named `sm1` and `sm2`. The arguments sent to the constructor specify the initial (x, y) locations of the snowmen, which the constructor stores in the two data members of the objects. Lines 19 and 20 of the client code invoke the `show` method. During the first execution of the method, the variables x and y on lines 27–31 of Figure 3.25 refer to the `x` and `y` data members of snowman `sm1`. Because these variables contain the coordinates (5, 30), this snowman is drawn in the upper left corner of the game board. Similarly, during the second invocation of the `show` method of the `SnowmanV3` class, the variables x and y on lines 26–31 refer to the `x` and `y` data members of snowman `sm2`, and it is drawn at the lower right corner of the game board (460, 423).

3.6.3 Private Access and the set/get Methods

In the interest of simplicity, the data members and the member methods of the classes we have discussed all had public assess as indicated by the plus (+) sign that precedes them in their UML diagrams. Another type of access available in Java is private access, which is denoted in a UML diagram by a minus (-) sign. In this section, we will examine the difference between public and

```
1   import java.awt.Color;
2   import java.awt.Graphics; //needed for drawing shapes
3
4   public class SnowmanV3
5   {
6     //data members
7     public int x = 5;
8     public int y = 30;
9     public Color hatColor = Color.BLACK;
10
11    // member methods
12    public SnowmanV3(int x, int y)
13    {
14      this.x = x;
15      this.y = y;
16    }
17
18    public void showXYToSC()
19    {
20      System.out.println("x is: " + x +
21                         "\ny is: " + y);
22    }
23
24    public void show(Graphics g) // g is passed to the method
25    {
26      g.setColor(hatColor);
27      g.fillRect(x + 15, y, 10, 15); //hat
28      g.fillRect(x + 10, y + 15, 20, 2); //brim
29      g.setColor(Color.WHITE);
30      g.fillOval(x + 10, y + 17, 20, 20); //head
31      g.fillOval(x, y + 37, 40, 40); //body
32    }
33  }
```

Figure 3.25
The class **SnowmanV3**.

```
import edu.sjcny.gpv1.*;
import java.awt.Graphics;

public class ConstructorAndThis extends DrawableAdapter
{
  static ConstructorAndThis ga = new ConstructorAndThis();
  static GameBoard gb = new GameBoard(ga,"Constructors and " +
                                        "Key Word:this");
  static SnowmanV3 sm1 = new SnowmanV3( 5, 30);
  static SnowmanV3 sm2 = new SnowmanV3(460, 423);

public static void main(String[] args)
{
  showGameBoard(gb);
}

public void draw(Graphics g) //the drawing call back method
{
  sm1.show(g);
  sm2.show(g);
}
}
```

Figure 3.26
The application **ConstructorAndThis**.

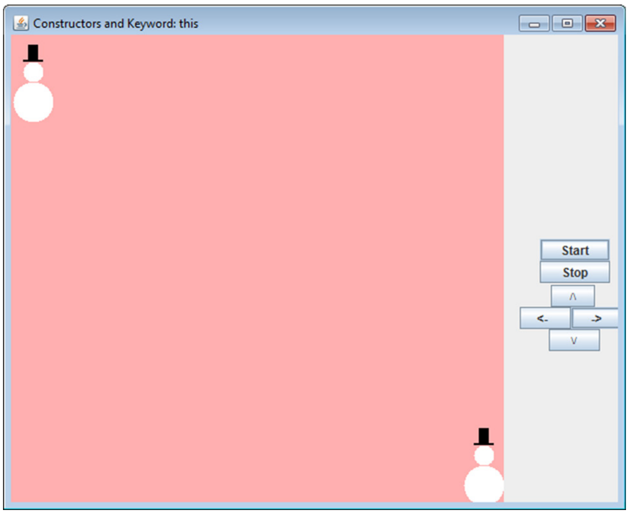


Figure 3.27
The output from the application **ConstructorAndThis**.

private access and learn which access modifier is normally used for the data members and member methods of a class. This will lead us to a discussion of methods that begin with the prefixes `set` and `get` that are commonly coded in most classes.

Public and Private Access

In the case of a member method, *access* is the act of invoking the method. In the case of a data member, access is the act of fetching or assigning the contents of the data member. Designating *private* access to the data members or methods of a class places no restrictions on the code of the methods contained in the class. Any line of code in a method of a class can invoke any private or public method in the class and can fetch and assign the value stored in any of its data members.

Private access places restrictions on client code. Client code cannot invoke methods of a class that are assigned private access, nor can it access the data members of a class that are assigned private access. If an application declared an object named `mary`, and the object's class contained a method named `show`, then the client code statement

```
mary.show();
```

would result in a translation error if the method was assigned private access.

NOTE *Public access allows client code to access a class's data member or method, but private access does not.*

The syntax we have used in client code to access (invoke) a public method can also be used in client code to access an object's public data. The syntax is the member (method or data) name proceeded by the object name followed by a dot. This syntax was used on line 12 of the application shown in Figure 3.21

```
sm1.showXYToSC()
```

to invoke the public method `showXYToSC` coded on lines 12–16 of Figure 3.20. This method outputs two data members, `x` and `y`, to the system console. Because the access modifier used in the declaration of these two data members is public, their contents could have been fetched and then output by the client code by replacing line 12 of the client code (Figure 3.21) with the statement:

```
System.out.println("x is: " + sm1.x + "\ny is: " + sm1.y);
```

In addition, the client code could set the x location of snowman `sm1` to 10 by coding:

```
sm1.x = 10;
```

Normally, methods in a class are assigned public access. Exceptions to this will be given in subsequent chapters. Assigning them public access allows the client code to invoke them.

Good programming practice dictates that all data members in a class be assigned private access because allowing the client public access to an object's data members can lead to some insidious and difficult to find programming errors.

That being said, it is often the case that client code has a need to obtain (get) the value stored in an object's private data member, or set the value to a new value. Because any method in a class can access both private and public data members that are part of its class, public methods that begin with the prefixes `get` and `set` are added to the class. The client then invokes these methods to fetch and change the values stored in an object's private data members.

NOTE *Normally, methods in a class are assigned public access, and data members are assigned private access. Client code invokes the* `set` *and* `get` *methods of the class to access an object's private data.*

Set Methods

A `set` method is a `void` method used to change, or set, the value stored in an object's private data member to a new value. The new value is passed into it as an argument. Because most classes have more than one data member, `set` is not a method name, but a prefix used in naming methods. Normally, we code a `set` method for every private data member in the class. The method names begin with the `set` prefix, which is followed by the name of the data member they operate on. For example, `setX` would be the name of the method the client code would invoke to change the contents of an object's private data member named `x`.

The signature of a `set` method contains one parameter and its code block contains one line of code. The method's parameter receives the new value of the data member, and the line of code simply assigns the new value to the data member. Because the value passed into the method is to be the new value of the data member, the parameter's type always matches the type of the data member. Below is the code of the `setX` method that sets the value of a private integer data member named `x` to the value of the argument passed to it.

```
public void setX(int x)
{
   this.x = x;
}
```

Because the method is public, the client code could invoke it to change the data member `x` of the object `sm1` to `100`:

```
sm1.setX(100);
```

String Immutability

The `String` class does not contain `set` methods to change the value of the characters stored in a `String` object. This is because Java strings are *immutable*. Once a value has been stored in a `String` object, the value cannot be changed. Although the following code fragment appears to change the value `"Robert"` stored in the string object created on the first line to `"Bob"`, in fact it does not. Rather, it creates a new string object, stores `"Bob"` in it, and assigns the address of the newly created object to the variable `name`.

```
String name = "Robert";
name = "Bob";
```

Although this gives the appearance that the value stored in the object has changed, in reality, the new string value `"Bob"` is stored in a different object. The process is illustrated in Figure 3.28.

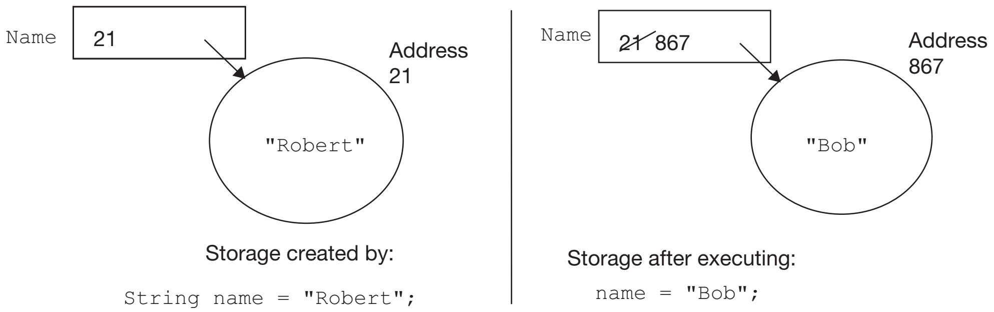


Figure 3.28
The immutability of String objects.

`get` Methods

A `get` method is a nonvoid method used to fetch, or get, the value stored in an object's private data member. The value is returned via a return statement. Like `set`, `get` is a prefix used in naming methods that fetch and return private data, and normally, a `get` method is coded for every private data member in a class.

Good programming practice dictates that the names of these methods begin with the prefix `get`, which is followed by the name of the data member on which they operate. For example, `getX` would be the name of the method the client code would invoke to fetch the contents of an object's private data member named `x`.

The signature of a `get` method contains a return type, and its parameter list is empty. The return type is always the same as the type of the data member it fetches. Its code block contains one line of code that simply returns the value of the data member. Below is the code of the `getX` method that returns the integer value of an object's private data member named `x`:

```
public int getX()
{
   return x;
}
```

Assuming a client application had declared an object named `sm1`, the following client code increases the object's private data member `x` by one:

```
int currentX = sm1.getX();
sm1.setX(currentX + 1);
```

Figure 3.29 shows the code of the class `SnowmanV4`. It is the same code as the class `SnowmanV3` shown in Figure 3.25, except its three data members have been assigned private access (lines 6–8) and the console output method has been removed. In addition, `set` and `get` methods for its private data members `x` and `y` have been added to the class. The code of these four methods, `getX`, `setX`, `getY`, and `setY` begin on lines 27, 32, 37, and 42, respectively.

```
import java.awt.*;

public class SnowmanV4
{
  //data members
  private int x = 7;
  private int y = 30;
  private Color hatColor = Color.BLACK;

  // member methods
  public SnowmanV4(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public void show(Graphics g) // g, is passed to the method
  {
    g.setColor(hatColor);
    g.fillRect(x + 15, y, 10, 15); //hat
    g.fillRect(x + 10, y + 15, 20, 2); //brim
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y + 17, 20, 20); //head
    g.fillOval(x, y + 37, 40, 40); //body
  }

  public int getX()
  {
    return x;
  }

  public void setX(int newX)
  {
    x = newX;
  }

  public int getY()
  {
    return y;
  }

    public void setY(int newY)
  {
    y = newY;
  }
}
```

Figure 3.29
The class **SnowmanV4**.

The application class `SetGetButtonClick` shown in Figure 3.30 illustrates the use of `set` and `get` methods to access private data members.

```
1    import edu.sjcny.gpv1.*;
2    import javax.swing.*;
3    import java.awt.Graphics;
4
5    public class SetGetButtonClick extends DrawableAdapter
6    {
7      static SetGetButtonClick ga = new SetGetButtonClick ( );
8      static GameBoard gb = new GameBoard(ga,"Get Set and Button Click");
9      static SnowmanV4 sm1 = new SnowmanV4(5, 30); //top-left corner
10    static SnowmanV4 sm2 = new SnowmanV4(460, 423); //bottom-right corner
11
12     public static void main(String[] args)
13     {
14       String s = JOptionPane.showInputDialog("sm2's new x location?");
15       int newX = Integer.parseInt(s);
16       sm2.setX(newX);
17       showGameBoard(gb);
18     }
19
20     public void draw(Graphics g) //the drawing call back method
21     {
22       sm1.show(g);
23       sm2.show(g);
24     }
25
26     public void rightButton() //moves sm1 one pixel right per click
27     {
28       int currentX = sm1.getX( );
29       sm1.setX(currentX + 1);
30     }
31   }
```

Figure 3.30
The application **`SetGetButtonClick`**.

The program is identical to the `ConstructorAndThis` application shown in Figure 3.26, except that lines 14–16 and 26–30 have been added. Lines 16 and 29 illustrate the use of the `setX` method to change the `x` value of Snowman `sm2` and thus reposition it horizontally. Lines 26–30 illustrate the use of the `getX` method and the `rightButton` call back method to move snowman `sm1` to the right one pixel every time the right button is clicked.

When the program begins, two snowmen, `sm1` and `sm2`, declared on lines 9 and 10, are displayed on the game board by lines 22 and 23 of the `draw` call back method (Figure 3.30) at the upper-left and lower-right corners of the game board. Then, line 14 displays an input dialog box asking the user to enter the new value of snowman `sm2`'s x coordinate. Line 16 invokes the `setX` method to operate on snowman `sm2`, passing it the new x coordinate parsed on line 15. The method stores the new value in `sm2`'s x data member (line 34 of Figure 3.29). The result is that when the game board is redrawn after the dialog box closes, the snowman is drawn at its new x position. Figure 3.31 shows snowman `sm2` in its new location, at the middle of the game board, after the user enters `250` in the input dialog box.

Lines 26–30 is an implementation of the game environment's `rightButton` call back method. As its name implies, this method is invoked every time the button on the game window with the right arrow head (→) is clicked. It uses the `getX` method on line 28 to fetch the current x coordinate of `sm1`. Then it invokes the `setX` method to set the x data member of snowman `sm1` to one more than its current value. Because the `rightButton` call back method executes every time the right button is clicked, and the `draw` method is invoked when it completes its execution, `sm1` moves one pixel to the right very time the button is clicked. The upper portion of Figure 3.31 shows `sm1`'s new location after 60 clicks of the game board's right (→) button.

3.6.4 The `toString` and `input` Methods

The methods we have developed in Section 3.6 perform work for a client application. They construct objects, display objects, and access the values of an object's private data members. The methods `toString` and `input` are two other methods we can write to perform work for the client applications. These methods expand the client's ability to access private data members, and both of the methods normally permit access to all of an object's data members in one invocation.

The `toString` Method

The `toString` method is a nonvoid method that returns a string containing all the annotated values of an object's data members to the client application. The method's parameter list is empty. Its code progressively concatenates identifying annotation with the value of each of an object's data members, and the resulting string is the method's returned value. For example, the `SnowmanV4` class shown in Figure 3.29 contains three data members: `x`, `y`, and `hatColor`. A typical coding of this class's `toString` method would be:

```
1    public String toString()
2    {
3       String s;
4       s = "x is: " + x +
5           "\ny is: " + y +
6           "\nhatColor is: " + hatColor;
7       return s;
8     }
```

Because a class can have an unlimited number of data members, it is good coding practice to code the concatenation of each data member's annotation and variable name on a separate line, as coded on lines 4, 5, and 6. Because the client code often sends the returned string to an output device, the inclusion of a new-line escape sequence in all but the first data member's annotation improves the readability of the output.

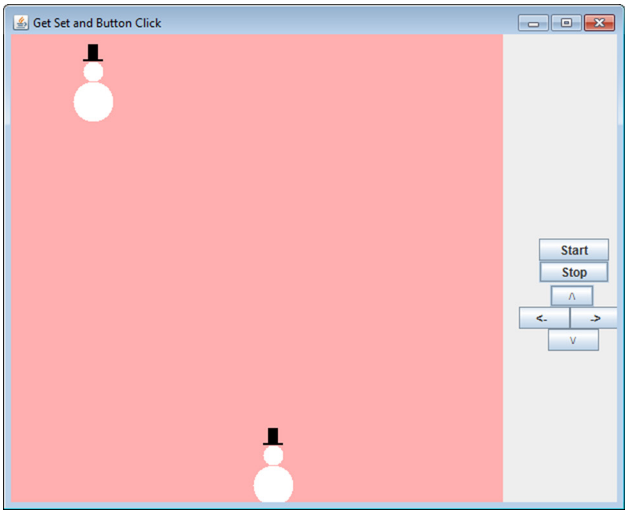


Figure 3.31
The output of the application **`SetGetButtonClick`** after the snowmen are relocated.

The variable `hatColor`,coded at the end of line 6, is declared as a reference variable in the `SnowmanV4` class. When a reference variable is coded where a string is expected (as is expected here because the variable is preceded by the concatenation operator), the translator considers it to be an implicit invocation of another `toString` method. In Section 3.5.4, the location stored in the reference variable `mary` was output with an explicit invocation of a Java-provided default `toString` method. Because the class `Color` contains its own `toString` method, that method is implicitly invoked on line 6, and the string it returns is then concatenated into the string `s`. Rather than returning contents of the variable `hatColor` (an address), `Color`'s `toString` method places a description of the color stored in the `Color` object `hatColor` into the returned string.

The `input` Method

The input method is a `void` method that allows the program user to enter new values for all of the data members of an object. The method's parameter list is empty. Its code prompts the user to enter new values for each of an object's data members, parses numeric inputs, and assigns the new values to the object's data members. For example, the `SnowmanV4` class shown in Figure 3.29 contains three data members: `x`, `y`, and `hatColor`. A typical coding of this class's `input` method would be:

```
public void input()
{
   String s;
   int red, green, blue;

   s = JOptionPane.showInputDialog("enter the value of x");
   x = Integer.parseInt(s);
   s = JOptionPane.showInputDialog("enter the value of y");
   y = Integer.parseInt(s);
```

```
10        s = JOptionPane.showInputDialog("enter hat's red intensity");
11        red = Integer.parseInt(s);
12        s = JOptionPane.showInputDialog("enter hat's green intensity");
13        green = Integer.parseInt(s);
14        s = JOptionPane.showInputDialog("enter hat's blue intensity");
15        blue = Integer.parseInt(s);
16        hatColor = new Color(red, green, blue);
17    }
```

Because the variables `x`,`y`, and `hatColor` are not declared within the method, assignments into them (on lines 7, 9, and 16) change the values stored in the `SnowmanV4` object's data members.

Line 16 creates a new color object using the `Color` class's three-parameter constructor and stores its address in the data member `hatColor`. The arguments sent to the constructor are the shade intensities of the colors red, green, and blue that combine to produce the desired new color. The range of a color's intensity is 0 (lowest intensity) to 255 (highest intensity). High intensities produce bright colors. The program user would have to have knowledge of how to mix shade intensities of these three colors to produce a desired color. These intensities are input and parsed on lines 10–15. In the simplest case, if the desired color were to be either red, green, or blue, the intensity of the other two colors would be input as zero. White is an equal mix of the three colors, and black is the absence (zero intensity) of the three colors.

Figure 3.32 presents the class `SnowmanV5` that includes the code of the `toString` (lines 29–36) and `input` methods (lines 38–54) discussed in this section, and Figure 3.33 presents the application `ToStringAndInput` that demonstrates the use of these methods. The console and graphical outputs produced by the program are presented in Figure 3.34.

Lines 8 and 9 of the application (Figure 3.33) declares two snowmen, `sm1` and `sm2`, located at (7, 30) and (460, 420), respectively. The `SnowmanV5` class's `toString` method is invoked inside of the `println` method's argument list on lines 13 and 14 to obtain annotated versions of the current values of each snowman's data members. The returned string is concatenated with the names of the snowman and output to the system console (top of Figure 3.34).

The output contains a description of each the snowman's current hat color: *java.awt.Color[r=0,g=0,b=0]*. This is the string returned from the `SnowmanV5` class's `toString` method's implicit invocation of the `Color` class' `toString` method (line 34 of Figure 3.32). The *r=0, g=0, b=0* portion of the output indicates that the red (r), green (g), and blue (b) intensities of the color are all zero: the default hat color black.

Line 15 of the application displays the game board, with the two snowmen drawn on it at their initial locations wearing their black hats (Figure 3.34a). The `SnowmanV5` class's `input` method is invoked on lines 20 and 21 of the application (Figure 3.33), which allows the user to input new values of the two snowmen's data members. Finally, line 18 redisplays the game board, and the two snowmen are drawn at their new locations with their new colored hats as shown on in Figure 3.34b. This output reflects user inputs of:

(200, 200) for `sm1`'s location and (0, 255, 0) for its (red, green, blue) color intensities;
(250, 200) for `sm2`'s location and (0, 0, 255) for its (red, green, blue) color intensities.

```
import java.awt.*;
import javax.swing.*; // needed for dialog box input

public class SnowmanV5
{
  //data members
  private int x = 7;
  private int y = 30;
  private Color hatColor = Color.BLACK;

  //member methods
  public SnowmanV5(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public void show(Graphics g) //g is passed to the method
  {
    g.setColor(hatColor);
    g.fillRect(x + 15, y, 10, 15); //hat
    g.fillRect(x + 10, y + 15, 20, 2); //brim
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y + 17, 20, 20); //head
    g.fillOval(x, y + 37, 40, 40); //body

  }

  public String toString()
  {
    String s;
    s = "x is: " + x +
        "\ny is: " + y +
        "\nhatColor is: " + hatColor;
    return s;
  }

  public void input()
  {
    String s;
    int red, green, blue;

    s = JOptionPane.showInputDialog("enter the value of x");
    x = Integer.parseInt(s);
    s = JOptionPane.showInputDialog("enter the value of y");
    y = Integer.parseInt(s);
    s = JOptionPane.showInputDialog("enter hat's red intensity");
    red = Integer.parseInt(s);
    s =JOptionPane.showInputDialog("enter hat's green intensity");
```

```
      green = Integer.parseInt(s);
      s = JOptionPane.showInputDialog("enter hat's blue intensity");
      blue = Integer.parseInt(s);
      hatColor = new Color(red, green, blue);
    }
}
```

Figure 3.32
The class **SnowmanV5**.

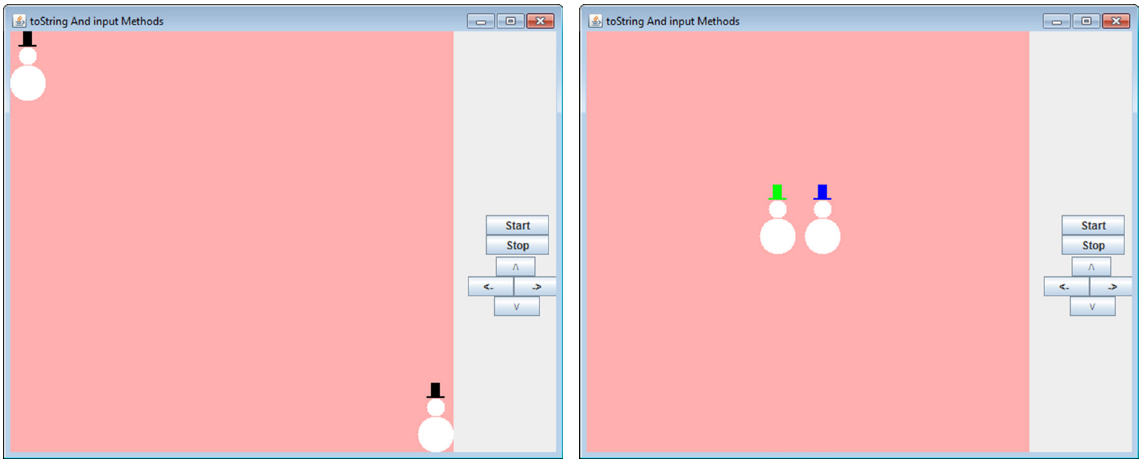
```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class ToStringAndInput extends DrawableAdapter
{
  static ToStringAndInput ge = new ToStringAndInput();
  static GameBoard gb = new GameBoard(ge,"toString And input
                                        Methods");
  static SnowmanV5 sm1 = new SnowmanV5(7, 30);
  static SnowmanV5 sm2 = new SnowmanV5(460, 420);

  public static void main(String[] args)
  {
    System.out.println("sm1's\n" + sm1.toString());
    System.out.println("sm2's\n" + sm2.toString());
    showGameBoard(gb);
    sm1.input();
    sm2.input();
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    sm1.show(g);
    sm2.show(g);
  }
}
```

Figure 3.33
The application **ToStringAndInput**.

Console Output:

sm1's
x is: 7
y is: 30
hatColor is: java.awt.Color[r=0,g=0,b=0]
sm2's
x is: 460
y is: 420
hatColor is: java.awt.Color[r=0,g=0,b=0]

Game Board Output:



(a) (b)

Figure 3.34
The console and game board output from the application **`ToStringAndInput`**.

3.7 OVERLOADING CONSTRUCTORS

Overloading constructors is an object oriented programming term used to describe a class that contains more than one constructor method. The code of each constructor is different, which is the motivation for coding more than one constructor. There is no limit on the number of constructors a class can contain. The name of a constructor method must be the name of the class, so all of the constructor methods in a class have the same name. For example, if the class's name is `SnowmanV4`, then the name of all of the constructors would be `SnowmanV4`.

Any of a class's constructors can be used by a client application to allocate an object in the constructor's class. Because the names of the methods are the same, the only way the Java translator knows which constructor is being used is to examine the type and number of arguments in the client's invocation statement.

Consider the code of the class `SnowmanV6` presented in Figure 3.35. It contains three constructors, which begin on lines 12, 15, and 20. The signature (line 12) of the first of these constructors contains no parameters and is therefore referred to as the no-parameter constructor. To use the no-parameter constructor, the client's declaration statement would not contain any arguments:

```
SnowmanV6 s1 = new SnowmanV6();
```

Because the constructor's code block is empty, the snowman's `x`, `y`, and `hatColor` data members would retain their default values set on lines 7–9, and when the snowman was drawn it would appear at (7, 30) with a black hat.

To use the two-parameter constructor on line 15, the client's declaration statement would have to contain two integer arguments:

```
SnowmanV6 s1 = new SnowmanV6(250, 250);
```

This constructor allows the client to specify the initial location of the newly created snowman. Lines 17–18 would execute and set the value copied into the method's parameters (250) into the object's x and y data members. When the snowman was drawn, it would appear at (250, 250) wearing a black hat.

To use the three-parameter constructor on line 20, the client's declaration statement would have to contain two integer arguments and a reference to a `Color` object:

```
SnowmanV6 s1 = new SnowmanV6(350, 250, Color.BLUE);
```

This constructor allows the client not only to specify the initial location of the snowman, but also its hat color. Lines 22–24 would execute and set the values 350 and 250 into the object's `x` and `y` data members, and it would set the object's data member `hatColor` to blue. In this case, when the snowman was drawn, it would appear at (350, 250), and because line 28 of Figure 3.35 uses the object's `hatColor` data member to set the current color before drawing the hat, it would be wearing a blue hat.

An attempt to create a snowman with an argument list that does not match one of the parameter lists on lines 12, 15, or 20 would result in a translation error. For example, the client statement

```
SnowmanV6 s1 = new SnowmanV6(350.34, 200, Color.BLUE);
```

would result in a translation error indicating that the translator can find a constructor whose parameters are a double, followed by an integer, followed by a `Color` object.

NOTE *Each constructor in a class must have a unique parameter list, and the type and number of the arguments in the client's object declaration statement must match one of these lists.*

It should be noted that once a constructor is coded in a class, the Java-provided default constructor discussed in Section 3.5.3 can no longer be used to create an instance of the class. As a result, to default to the values in data member's declaration statements, a no-parameter constructor (e.g., lines 12–14 of Figure 3.35) must be added to the class.

Figure 3.36 presents the application `OverloadingConstructors` that uses the three constructors shown in Figure 3.35 to construct three snowmen on lines 8, 9, and 10: one at the default location (7, 30), one at the center of the game board (250, 250), and one to its right at (350, 250) with a blue hat. Before each snowman's hat is drawn, line 28 of the snowman's class's show method (Figure 3.35) sets the current drawing color to the snowman's hat color. As a result, when the three snowmen are drawn on the game board (lines 19–21 of Figure 3.36) at their initial locations, one is wearing a blue hat (Figure 3.37).

```
import java.awt.Color;
import java.awt.Graphics; // needed for drawing shapes

public class SnowmanV6
{
  //data members
  private int x = 7;
  private int y = 30;
  private Color hatColor = Color.BLACK;

  // member methods
  public SnowmanV6( )
  {
  }
  public SnowmanV6(int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  public SnowmanV6(int x, int y, Color hatColor)
  {
    this.x = x;
    this.y = y;
    this.hatColor = hatColor;
  }
  public void show(Graphics g)   // g is passed to the method
  {
    g.setColor(hatColor);
    g.fillRect(x + 15, y, 10, 15);      // hat
    g.fillRect(x + 10, y + 15, 20, 2);  // brim
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y + 17, 20, 20); // head
    g.fillOval(x, y + 37, 40, 40);      // body
  }
  public int getX()
  {
    return x;
  }
  public void setX(int newX)
  {
    x = newX;
  }
  public int getY()
  {
    return y;
  }

  public void setY(int newY)
  {
```

```
        y = newY;
    }
}
```

Figure 3.35
The class **SnowmanV6**.

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class OverloadingConstructors extends DrawableAdapter
{
  static OverloadingConstructors ga= new OverloadingConstructors();
  static GameBoard gb = new GameBoard(ga, "Overloading
                                          Constructors");
  static SnowmanV6 sm1 = new SnowmanV6( 7, 30);
  static SnowmanV6 sm2 = new SnowmanV6 (250, 250);
  static SnowmanV6 sm3 = new SnowmanV6(350, 250, Color.BLUE);

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }

  public void draw(Graphics g) //the drawing call back method
  {
    sm1.show(g);
    sm2.show(g);
    sm3.show(g);
  }
}
```

Figure 3.36
The application **OverloadingConstructors**.

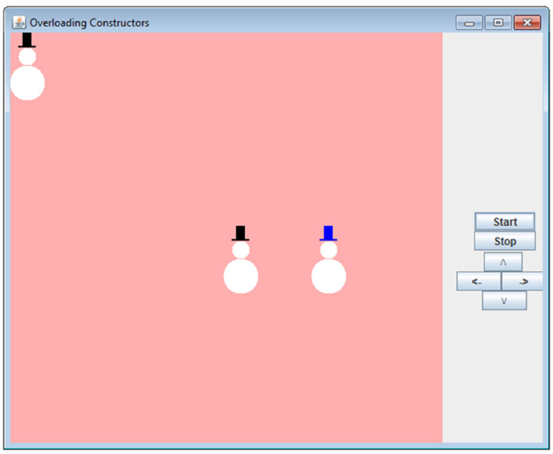


Figure 3.37
The output from the application **OverloadingConstructors**.

3.8 PASSING OBJECTS TO AND FROM WORKER METHODS

The techniques and syntax used to pass primitive information (i.e., values stored in primitive variables) between client and worker methods were discussed in Section 3.2. The same techniques and syntax presented in that section can be used to pass objects between client and worker methods. In the case of objects, the information passed is actually the addresses of the objects stored in the reference variables that refer to the objects. An unlimited number object addresses can be passed to a worker method via its parameter list, and the address of one object can be returned from a worker method via its `return` statement.

Passing Objects to Worker Methods

The first row of Table 3.3 gives the syntax used to invoke the `Game` class's *static* method `add1ToX` passing it the address of the `SnowmanV6` object `sm1`. The right-most column gives the syntax of the method's signature. For comparative purposes, the second row of the table gives the syntax used to pass the integer `age` to static method `add1toAge` and the syntax of the method's signature. As shown in the table, the syntax used to pass objects to worker methods is the syntax used to pass primitive values to worker methods. The primitive type coded in the method's parameter list is replaced with the type of the reference variable (i.e., the object's class name), as shown in the rightmost column of the table.

The following code segment is a `static` worker method named `moveRight` that increases the `x` data member of the `SnowmanV6` object passed to it by one pixel:

```
public static void moveRight(SnowmanV6 aSnowman)
{
 int currentX = aSnowman.getX( );
 aSnowman.setX(currentX + 1);
}
```

Table 3.3
Syntax Used To Pass Objects and Primitives to Worker Methods

Information Passed	Client Method's Invocation Statement	Worker Method's Signature Coded in the Class Game
An Object's address	`Game.add1ToX(sm1)`	`static void add1ToX(SnowmanV6 sm)`
An Integer value	`Game.add1ToAge(age1)`	`static void add1ToAge(int age)`

Figure 3.38 is modified version of the program presented in Figure 3.30 that moves a snowman one pixel to the right every time the game board's right arrow button is clicked. The method `moveRight` has been added to the program (lines 32–37), and it is used to move two snowmen to the right every time the right arrow button is clicked. This method is invoked on lines 28 and 29 to move the two `SnowmanV6` objects, `sm1` and `sm2`, to the right one pixel. The objects are created on lines 9 and 10 using the class's three-parameter constructor. The first invocation of `moveRight` (line 28) passes the location of `sm1` to the method, and the second invocation (line 29) passes `sm2`'s location to the method. Because the static method `moveRight` is coded in the same class as the invocation statements on lines 28 and 29, the name of the class need not be included in the invocations.

```java
import edu.sjcny.gpv1.*;
import javax.swing.*;
import java.awt.*;

public class ObjectsAsParameters extends DrawableAdapter
{
  static ObjectsAsParameters ga = new ObjectsAsParameters();
  static GameBoard gb = new GameBoard(ga, "Objects As Parameters");
  static SnowmanV6 sm1 = new SnowmanV6(5, 40, Color.RED);
  static SnowmanV6 sm2 = new SnowmanV6(460, 423, Color.BLUE);

  public static void main(String[] args)
  {
    String s = JOptionPane.showInputDialog("sm2's new x location?");
    int newX = Integer.parseInt(s);
    sm2.setX(newX);
    showGameBoard(gb);
  }

  public static void draw(Graphics g) // the drawing call back method
  {
    sm1.show(g);
    sm2.show(g);
  }

  public void rightButton() //moves sm1 & sm2 one pixel right per click
  {
    moveRight(sm1);
    moveRight(sm2);
  }

  public void moveRight(SnowmanV6 aSnowman)
  {
    int currentX = aSnowman.getX( );
    currentX++;
    aSnowman.setX(currentX);
  }
}
```

Figure 3.38
The application **`ObjectsAsParameters`**.

Figure 3.39 illustrates the passing of the location from the reference variable `sm1` into the method's `moveRight` parameter `aSnowman`, and the change in the `x` data member of the object after the method executes. The client code's RAM memory is shown on the left side of the figure, and the worker method's RAM memory is shown on the right side of the figure. Each time the method is invoked, the value stored in the invocation's argument is copied into the parameter

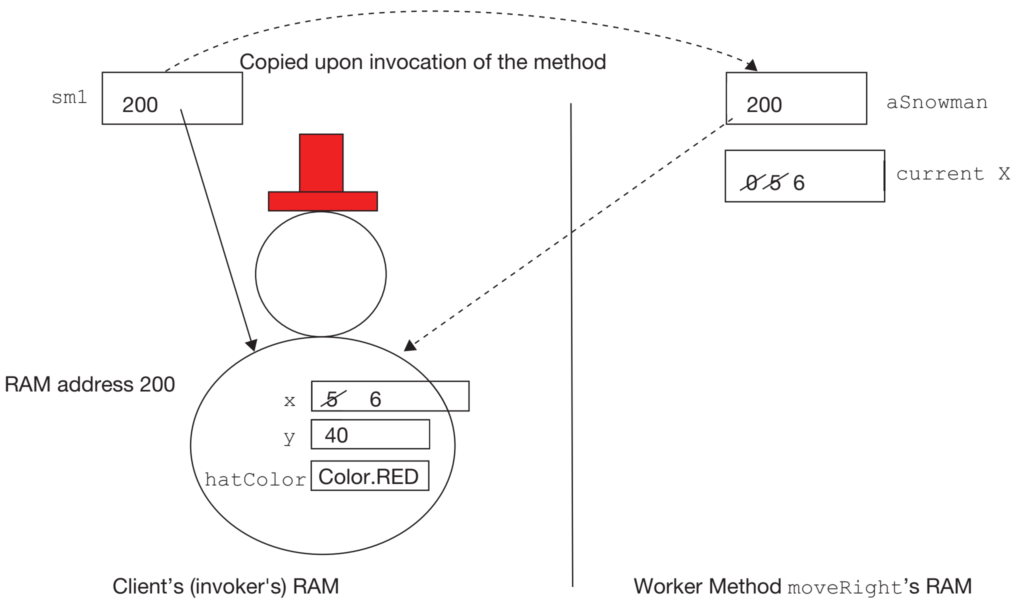


Figure 3.39
The passing of the object sm1 to the worker method `moveRight`.

`aSnowman`. The dashed arrow at the top of the figure illustrates this process for the first invocation of the method (line 28) when the location of snowman `sm1` is passed to the parameter `aSnowman`.

After the snowman's location, 200, is copied into the worker method's parameter `aSnowman`, the use of this variable on lines 34 and 36 of Figure 3.38 refers to the client's snowman object `sm1`. Line 34 fetches `sm1`'s `x` data member, and line 36 changes the value stored in this data member. While the method is in execution, the snowman object is shared between the client code and the worker method it invoked. Although we normally say we are "passing an object to a method," we really should say we are "passing the address of the object to a method."

NOTE *Technically speaking, objects are not passed to and from methods. Rather, the addresses of the objects are passed between the methods.*

Because the `sm1`'s address is shared, when the worker method ends the initial value of its `x` data member (10) stored inside the object has been overwritten with the value 11. This is not a contradiction of the idea that value parameters prevent worker methods from changing the client's information passed to it as parameters because the information passed to the method `moveRight` is the contents of the variable `sm1`, not the object's data member `x`. This is a subtle but important point to understand. While it is true that the worker method can change the contents of the data members of the object `sm1` because `aSnowman` stores the object's address, it cannot change the address stored in the variable `sm1` (which was passed to it).

Returning an Object from a Worker Method

An object's address can be returned from a method using the same syntax used to return a primitive value from a method. The keyword **`void`** in the method's signature is replaced with the type of the information being returned. To return the location of an object from a method, the name of the returned object's class replaces the keyword **`void`**. As is the case when primitive values are returned from a method, if the returned address is to be used by the client code that invoked the method, the client code must assign the returned address to a variable.

The static method shown in Figure 3.40 creates a snowman object located half way between the two snowmen whose addresses are passed into its parameters, and returns the address of the newly created snowman. Assuming the method is added to the class `SnowmanV6`, the following code fragment invokes the method and stores the returned address of the newly created snowman in the reference variable `aSnowman`:

```
SnowmanV6 aSnowman;
aSnowman = SnowmanV6.halfWayBetween(snowman1, snowman2);
```

The signature of the method on line 1 of Figure 3.40 states that the address of a `SnowmanV6` object will be returned from the method. A `SnowmanV6` object is created on line 4, and its address is returned on line 9.

```
1   public static SnowmanV6 halfWayBetween(SnowmanV6 sm1,SnowmanV6 sm2)
2   {
3     int x, y;
4     SnowmanV6 aSnowman = new SnowmanV6();
5     x = (sm1.getX() + sm2.getX()) / 2);
6     y = (sm1.getX() + sm2.getX()) / 2);
7     aSnowman.setX(x);
8     aSnowman.setY(y);
9     return aSnowman;
10  }
```

Figure 3.40
A method that returns an object.

3.9 CHAPTER SUMMARY

This chapter began our study of the concepts used to design and implement classes, which will be expanded in Chapters 7 and 8. We learned that a class is similar to a blueprint enabling us to define and construct an item, and that an object is a particular item or instance of the class. In the same manner that we use the classes and methods available in the Java API to facilitate the design and development of a program, we can also use and reuse the classes we create.

Methods are subprograms, which are key components of classes. They perform the work of the class by creating, displaying, and manipulating the class's objects. Several versions of a class's constructor methods are normally available in a class to create an object and initialize various subsets of its data members. The names of methods that perform tasks common to most classes have been standardized, and they are included in most classes. The methods named `toString` and `show` are used to display an object on the console or on the game board, and the `input` method and methods that whose names begin with the prefixes `set` and `get` are used to change the values of an object's data members.

The first line of a method is called the method's signature. Java uses value parameters to pass information to a method and a return statement to transfer one value from a method. The list of information passed to a method is called an argument list, which is a sequence of variables and literal values separated by commas. This information is copied into the list of variables declared in the method's signature, which is called a parameter list. Before the method begins execution, the value stored in the i^{th} argument of the invocation statement is copied into the ith parameter of the method. An argument's type must match the type of its corresponding parameter. Value parameters prevent a method from changing the value stored in thevariables coded in the argument list.

Several methods in a class can have the same name if their parameter lists are different. When this feature is used in the coding of a class's methods, we say that the methods with the same name are overloaded. Constructor methods are often overloaded because their names must be the same as the class's name. Normally, methods have public access to permit methods defined outside of the class to invoke them, and data members have private access to prevent methods defined outside of the class from inadvertently changing their values.

A class's data members are declared as class level variables. Class level variables are variables declared outside of the code block of a method and inside the code block of the class. It is good programming practice to declare these class-level variables at the beginning of the class's code block before the implementation of the class's methods.

All class variables declared in a class, whether they are declared public or private, can be directly accessed within the class's methods by simply coding the name of the variable. The only exceptions to this are if the method declares a parameter, or variable, within its code block with the same name. When this is the case, the class variable is accessed within the method by preceding its name with the key word `this` followed by a period. The context in which direct access syntax can be used to access a variable is called the scope of the variable.

A UML diagram is a graphical depiction of a class developed during the design of the class. This tool not only facilitates the design of the class, but it also documents the data members and methods that make up the class. It is used as the starting point for the implementation of the class. Other class design tools discussed in this chapter are the techniques used to depict and digitize a graphical object, which serve as the basis for the implementation of their `show` methods.

The methods in the API `Graphics` class can be used to implement a graphical object's `show` method. This class provides methods for drawing lines and basic shapes on a previously declared `Graphics` object. The units of the (x, y) location of the lines, shapes, and the size of the shapes passed to these methods is pixels or picture elements. These methods provide the foundation for the rest of the graphical topics in this text.

Knowledge Exercises

1. Indicate whether the following statements are true or false:
   a) Methods must be coded inside the code block (i.e., the open and close brackets) of a class.
   b) The first line of a method is called its title.
   c) The first line of a method always ends with a semicolon.
   d) All methods must contain a code block.
   e) The method `pow` in the `Math` class is an example of a void method.
   f) We can invoke methods we did not write.
2. Indicate whether the following statements are true or false.
   a) It is good programming practice to begin a method name with an uppercase letter.
   b) It is good programming practice to make the names of methods representative of the work they perform.
   c) A method's name should not contain capital letters.
3. Fill in the blank:
   a) The signature of a method that does not operate on an object must contain the keyword ___________.
   b) The signature of a method that does not return a value must contain the keyword ___________.
   c) When we invoke a static method, we begin the invocation statement with the name of ___________ followed by a dot.
   d) When we invoke a nonstatic method, we begin the invocation statement with the name of ___________followed by a dot.
4. Give the invocation statement to invoke the class `Boat`'s `moveBoat` method whose signature is: `public static void moveBoat()`.
5. Indicate whether the following statements are true or false:
   a) Client code is code that invokes a method.
   b) A method can invoke the same method more than once.
   c) Parameters are used to pass information to a method, and the information is passed into the method's arguments.
   d) One or more pieces of information can be passed to a method.
   e) One or more pieces of information can be returned from a method.
   f) The type of a parameter must match the type of the information it receives.
   g) Parameters and arguments share the same variable.
   h) Java passes information to methods using the concept of reference parameters.
   i) When a method changes the value of an integer passed to it, the original value is no longer available to the client code.
6. Give the signature of a public method named `add` that adds two integers sent to it and returns the result.

7. After an invoked method completes its execution, which statement executes next?

8. Indicate whether the following statements are true or false:
   a) A class-level variable must be coded inside a method in the class.
   b) Class-level variables are used to share information between all of the methods defined in the class.
   c) A method cannot declare a variable with the same name as a class-level variable.
   d) When a method changes the value stored in a class-level variable, the original value is no longer available to the other method in the class.
   e) More than one class-level variable can be coded in a class.

9. Give the declaration of a class-level variable named `checkAmount` that is coded in the program's class.

10. Fill in the blank:
    a) The method ___________ in the `Graphics` class is used to change the current drawing color.
    b) The constant___________in the `Color` class stores the color red.
    c) The import statement ___________ is used to access the methods defined in the `Graphics` class.
    d) The import statement ___________ is used to access the color constants defined in the `Color` class.

11. Give the name of the method in the `Graphics` class used to:
    a) Draw the outline of a rectangle
    b) Draw a filled rectangle
    c) Draw the outline of an ellipse
    d) Draw the outline of a circle
    e) Draw a filled circle
    f) Draw a line

12. Give the Java statement (or statements) to draw the following shapes and lines on the `Graphics` object `g`:
    a) A line from (200, 30) to (100, 75) drawn in the current color
    b) The outline of a 100-pixel wide by 50-pixel high rectangle located at (20, 200) drawn usingthe current color
    c) A blue filled circle whose diameter is 30 pixels located at (250, 300)
    d) A blue filled ellipse 100-pixel wide by 50-pixel high located at (300, 100)

13. Fill in the blank:
    a) Using the words object and class in: House is to ___________ as blueprint to___________.
    b) Classes are comprised of member ___________ and ___________.
    c) The name of the graphic used to specify a class is a ___________ diagram.

d) Data members of a class are usually designated to have ___________ access.

e) Member methods of a class are usually designated to have ___________ access.

14. Give the Java code to declare an object named `joe` in the class `Person` using the class's no-parameter constructor, and:

a) the one-line declaration syntax.

b) the two-line declaration syntax.

15. Referring to Exercise 14:

a) What is actually stored in the variable `joe`?

b) Is `joe` a primitive-type variable? If not, what is the its type?

c) Draw a picture (similar to Figure 3.13) of the memory allocated by Exercise 14a, assuming the class `Person` has two integer data members named `age` and `idNumber`.

16. Give the Java code to declare a class whose object will be coffee cups. Each coffee cup will have a size (ounces) and a price. The class will not contain any methods.

17. Referring to Exercise 16:

a) Give the code of the two-parameter constructor of the class defined in Exercise 16.

b) Give the client code used to declare a $3.85 coffee cup whose size is 8 ounces.

c) Give the code to output the coffee cup declared in part B to the system console using an implicit invocation of the `toString` method.

d) Repeat part C of this question using an explicit invocation of the `toString` method.

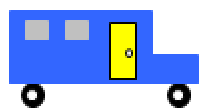
e) What is output to the console by the invocation in part C and B?

f) Give the code to produce the same output generated by part D to the graphic object `g`.

18. Give the code of a method named `toString` that, when added to the class defined in Exercise 16, returns the values of its two data members fully annotated.

19. Give the code of a method named `show` that, when added to the class defined in Exercise 16, outputs the values of its two data members to the center of a 500 wide by 500 high `Graphics` object named `g`.

20. Using a sketch similar to Figure 3.16, show the design of a recreational vehicle (RV) that has two side windows, tires a large entrance door.



21. Give a table similar to Table 3.2 that contains the digitized version of the RV design specified in Exercise 20.

22. When must the keyword **`this`** be used in a method to access one of the data members of its class?

23. A class contains one integer data member named `total` whose access is private.

a) Use the keyword **`this`** in a statement coded inside one of the class's method that doubles the value stored in the data member `total`.

b) Give a statement coded inside one of the class's method that doubles the value stored in the data member `total` without using the keyword **`this`**.

c) Give the code of a `set` method that client code could use to change the value of the data member `total`.

d) Give the code of a `get` method that client code could use to fetch the value of the data member `total`.

e) Assuming the appropriate `set` and `get` methods exist, give the client code to double the value of the total of the object named `myAccount` that uses the `set` and `get` methods.

f) Assuming the data member `total` was declared to have public access, give the client code to double the value of the total of the object named `myAccount` without using `set` and `get` methods.

g) Give the code of the class's `toString` method.

h) Give the code of the class's `input` method.

i) Which access modifier key word, **`public`** or **`private`**, results in more restricted access?

24. Indicate whether the following statements are true or false:

a) A class need not contain an explicitly coded constructor.

b) A class can contain several constructors.

c) A class can contain several constructors with different names.

d) A class can contain several constructors with the same signature.

e) When a constructor invocation is proceed by the keyword **`new`**, an object is created, and its address is returned.

f) The Java provided default constructor has no parameters.

25. A client application has declared three objects named `ship1`, `ship2`, and `ship3` that are instances of the existing class `Starship`. Each starship contains a data member that stores the color used to draw the starship.

a) Give the signature of a static method in the `Starship` class named `largest` that is passed two `Starship` objects and returns the address of one of them.

b) Give the client code statement used to invoke the static method described in part A of this exercise and place the returned address in `ship1`.

c) If the method changed the value of the color data member of one of the starships passed to it, would it be drawn in the new or old color after the method completes its execution?

d) Give the signature of a nonstatic method in the `Starship` class named `sameModel` that compares two `Starship` objects and returns a Boolean value.

e) Give the client code statement used to invoke the nonstatic method described in part D of this exercise and store the returned Boolean value in the variable `isSame`.

26. Using a sketch similar to Figure 3.16, show the design of the user-controlled game piece that is part of the game you specified in Preprogramming Exercise 1 of Chapter 1.

27. Give a table similar to Table 3.2 that contains the digitized version of the game piece design specified in Exercise 26.

Programming Exercises

1. Write a nongraphical application that contains a static method with an empty parameter list that outputs your name and your age on one line to the system console. The main method of the application should invoke it three times. The output it produces should be annotated as shown below (assuming your name is Tommy and you are 18 years old):

The author of this program is Tommy who is 18 years old.

2. Write a graphical application that contains a static method that is invoked by the `draw` call back method. It should have one parameter to receive the `Graphics` object `g` passed to it. When invoked, the method should output your name and your age to the center of the game board as shown below (assuming your name is Tommy and you are 18 years old):

The author of this program is Tommy who is 18 years old.

3. Write a nongraphical application that contains a static method to compute and return the square root of the product of three real numbers passed to it. The `main` method of the application should invoke it and then output the three numbers and the returned value clearly annotated.

4. Write a graphical application that contains a static method to compute and return the square root of the product of three real numbers passed to it. The `draw` method should invoke it and then output the three numbers and the returned value clearly annotated.

5. Write a nongraphical application that contains a static method to compute and return the square root of the product of three real numbers that are declared and initialized as class-level variables. The `main` method of the application should invoke it and then output the three numbers and the returned value clearly annotated.

6. Write a graphical application whose `draw` method displays an old television on a table with an antenna on it.

7. The statistics kept for each player on a ladies softball team include each player's name, number of homeruns, and batting average a real number.
   a) Give the UML diagram for a class named `TeamMember` whose objects can store the three private pieces of data for a player. The class should include a three-parameter constructor, a `toString` method, a method to input the statistics for a player, and a `showSC` method to output a player's statistics to the system console.
   b) Progressively implement and test the `TeamMember` class by adding one method and verifying it before adding the next method. A good order to add the methods is the `toString` method, followed by the constructor, the `showSC` method, and finally the `input` method. (The client code can create a `TeamMember` object using the Java supplied default constructor to test the `toString` method.)
   c) After all of the methods are verified, comment out the test code in the client application and add two `TeamMember` instances to the program whose statistics are passed to the three-parameter constructor. Output these players to the system console and then output them again after the user inputs new names, home run counts, and batting averages for each player.

8. Write a graphical application that contains a class named `RV` whose objects are the recreational vehicle designed and digitized as described in Knowledge Exercises 20 and 21. The class's private data members should be the vehicle's body color and (x, y) location.
   a) Give the UML diagram for the class. It should include a three-parameter constructor, a `toString` method, a method to input the values of all of an object's data members, and a `show` method to draw the RV at its current (x, y) location.

b) Progressively implement and test the `RV` class by adding a method and verifying it before adding the next method. A good order to add the methods to the class is the three-parameter constructor, followed by the `toString` method, the `show` method, and finally the `input` method. The client code should create an `RV` object using the three-parameter constructor to test all of the methods as they are progressively added to the class.

c) After all of the methods are verified, comment out the test code in the client application and add two `RV` instances to the program whose location and color are passed to the three-parameter constructor. Output these vehicles to the system console and the game board and then output them again after the user inputs a new color and a new (x, y) location for each vehicle.

9. After implementing and testing the class described in Programming Exercise 7, progressively add a `set` and a `get` method to the class for each of the class's data members. After the `set` and a `get` methods have been verified, create two instances of the class using the three-parameter constructor and display them to the system console. Then, ask the user how many home runs and batting average points should be added to each player's statistics. Use the `set` and a `get` methods to change the statistics and then output the two players to the system console.

10. After implementing and testing the class described in Programming Exercise 8, progressively add a `set` and a `get` method to the class for each of the class's data members. After the `set` and a `get` methods have been verified, create an instance of the class using the three-parameter constructor and display it on the system console and the game board. Every time one of the game board's directional buttons is clicked, the RV's location should be changed by two pixels in the appropriate direction.

11. Using the skills developed in this chapter, begin to implement the game you specified in Preprogramming Exercise 1 of Chapter 1. Begin by completing Knowledge Exercises 26 and 27 to design and digitize the user-controlled game piece. Then, implement the class of the digitized game object, beginning with a UML diagram of the game piece that includes a constructor with the appropriate number of parameters, a `show` method to draw the object on the game board at its current (x, y) location, a `toString` method, and a `set` and a `get` method for each of the class's data members. After progressively implementing and testing all of the class's methods, write a graphical application that displays the game piece on the game board and then moves the game piece by two pixels in the appropriate direction every time one of the game board's directional buttons is clicked.

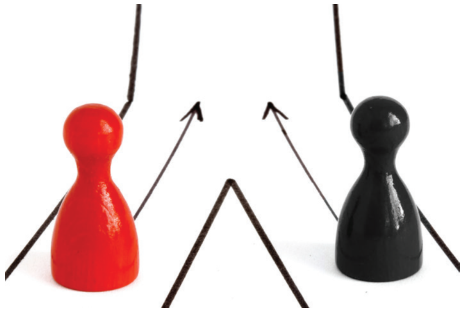
Endnotes and References

[1] Lanzinger, Franz. *Classic Game Design: From Pong to Pac-Man with Unity*. Dulles, Virginia: Mercury Learning and Information, 2014.

[2] Schell, Jesse. *The Art of Game Design*. Burlington, MA: Morgan Kaufmann Publishers, 2010.

CHAPTER 4

BOOLEAN EXPRESSIONS, MAKING DECISIONS, AND DISK INPUT AND OUTPUT



In this chapter

To control the sequence of operations, Java provides three decision-making statements, and in this chapter, we will learn how to write the Boolean condition on which these decisions are based. By default, Java statements execute in the order in which they are coded, although at some point in most algorithms, a decision has to be made as to which of its steps should be executed next. When depicted in a flow chart, this part of the algorithm begins with a diamond shape. To implement these algorithms, programming languages include decision statements that use Boolean expressions as conditions to determine whether to execute or skip statements. Java also provides two statements that always skip a predetermined set of statements, one of which will be discussed in this chapter.

In addition, this chapter extends the input and output techniques of the previous chapters to include input from the system console as well as disk I/O, and it introduces a technique used to alter the sequential execution path of a program when an unexpected error occurs.

After successfully completing this chapter, you should:

- Be familiar with the logical and relational operators and their order of precedence
- Be able to write and evaluate simple and complex Boolean expressions
- Understand how to compare strings and determine their alphabetic order
- Be able to write `if`, `if-else`, and `switch` statements to implement the decision-making part of an algorithm

- Understand the use of the `break` statement
- Be able to perform console input using the `Scanner` class and its methods
- Be able to create, open, read, and write sequential text files to and from a disk
- Begin to understand how to use `try` and `catch` blocks to handle an error exception
- Use decision-making statements to control a timer, a graphical object's visibility and motion, and detect collisions between two objects

4.1 ALTERNATIVES TO SEQUENTIAL EXECUTION

When a Java application begins, the first statement to execute is the first executable statement in the method main. The order in which the remainder of the instructions execute is referred to as the *execution path* of the program, or the *flow* of the program. The default execution path of a Java statement block is sequential. It can be thought of as the statements executing in line number order. After the first statement in the block executes, the remaining statements execute in the order in which they are written unless one of the statements specifically alters the execution path.

Many algorithms cannot be formulated in a way that all of its steps are executed sequentially. Therefore, programming languages provide statements to change the default sequential execution path. Programmers use these statements, or constructs, to alter the sequential flow of the program, so they are referred to as *control-of-flow* or *control* statements. We have already used one of these constructs: the invocation of a method. Assuming a method was invoked on line 10 of a program, the next statement to execute would not be line 11, but rather, the first executable statement in the method's code block. Line 11 would execute after the method completed its execution.

Aside from method invocation statements, programming languages provide additional control-of-flow statements to alter the default execution path. Some of these statements are used to skip a group of statements, and others are used to repeat a group of statements. Most often, these control-of-flow statements include a logical expression, called a Boolean expression, to decide when to skip statements or to decide how many times to repeat statements. In this chapter, we will discuss the Java statements used to skip a group of statements. The Java statements that are used to repeat a group of will be discussed in Chapter 5.

4.2 BOOLEAN EXPRESSIONS

Boolean (logical) expressions are named after George Boole, an English mathematician who conceived of a symbolic algebra for logic. Like mathematical algebraic expressions, Boolean expressions consist of operators and operands. Unlike mathematical expressions, Boolean expressions evaluate to one of two values: `true` or `false`. Boolean expressions used in control-of-flow statements can either be a *simple* Boolean expression, or a combination of two or more simple Boolean expressions called compound Boolean expressions.

4.2.1 Simple Boolean Expressions

A simple Boolean expression evaluates to either `true` or `false`. In Java, these expressions consist of a relational or an equality operator surrounded by two operands. Java's four relational operators are given at the top of the Table 4.1, and its two equality operators are at the bottom of the table. The first column in the table gives the name (which implies the meaning) of each operator, and the second column gives the Java symbols (keystrokes) that represent them. The symbols for the last four operators in the table are typed as two keystrokes without spaces. The third column of the table gives examples of simple Boolean expressions involving each of the six operators, all of which evaluate to `true`.

Table 4.1
Java's Relational and Equality Operators

Operator	Java Symbol	Examples that Evaluate to true
Less than	<	5 < 7
Greater than	>	6.31 > 3.14
Less than or equal to	<=	5 <= 5
Greater than or equal to	>=	'c' >= 'a'
Equal to	==	6 == 3 * 2
Not equal to	!=	23 != 54

! TIP

A common mistake made when coding simple Boolean expressions is to code the equal to operator as a single equal (=) keystroke, which is interpreted by the translator as the assignment operator. Think of this operator as "is equal" to (= =).

When one of the four relational operators is used, the two operands can be anything that can be converted to (interpreted as) a numeric. This includes numeric literals, numeric variables, and arithmetic expressions, as well as character literals and character variables. When a character literal or character variable is used, the character (e.g., `'A'`) is interpreted as an integer (e.g., 65), and the numeric value is used to evaluate the relational expression. The following code fragments are syntactically correct, and the third one evaluates to `true` because `'A'` and `'C'` are interpreted as 65 and 67 (see Appendix C), then the expression is evaluated.

```
int age = 13;
5 < 2 * 21
100 >= age
'A' < 'C'
25 <= 2 * (age + 1)
```

The interpretation of characters in simple Boolean expressions as numeric imposes an ordering on them called *lexicographical* or dictionary order, which is the order in which they appear in the Extended ASCII table (Appendix C).

When the types of the operands used with one of the four relational operators do not match (e.g., one is a float and one is a double, or one is an integer and the other is a character), one of the operands is promoted before the expression is evaluated. For example, the following simple Boolean expressions are syntactically correct and evaluate to `true`:

```
4.521 < 10         // 10 is promoted to the double 10.0
Math.PI >= -2      // -2 is promoted to double -2.0
2 < 'A'            // 'A' is promoted to 65
```

When one of the two equality operators is used in a simple Boolean expression, the choices for the operands are expanded. Not only can the two operands be anything that can be converted to a numeric, but they can also be two Boolean operands (literals or variables) or two reference variables (including the value `null`). For example:

```
4.535 != 21
65 == 'A'
true != false
myName != yourName
name == null
```

Like the arithmetic operators, the operators in Table 4.1 have an order of precedence associated with them. The four relational operators have equal precedence, and the two equality operators have equal precedence. The precedence value of the relational operators is higher than the precedence value of the equality operators. The expression

```
true == 'C' >= 'A'
```

is syntactically correct and evaluates to `true` because first `'C' >= 'A'` evaluates to `true`, and then `true == true` evaluates to `true`. As shown in Appendix E, the arithmetic operators have higher precedence than the relational and equality operators.

NOTE

Arithmetic expressions in simple Boolean expressions are evaluated first. In more complex expressions, the relational operators are evaluated next, followed by the equality operators, and then the logical operators. The assignment operator is evaluated last.

4.2.2 Compound Boolean Expressions

Like simple Boolean expressions, compound Boolean expressions also evaluate to either `true` or `false`. When used in a control-of-flow statement, compound Boolean expressions use the Java conditional binary logical operators AND and OR to combine the truth values of two or more operands. Alternately, compound Boolean expressions can use the unary logic operator NOT to reverse the truth value of a single operand, just as the negation operator reverses the sign of an operand in a mathematical expression. The operands in compound Boolean expressions must evaluate to Boolean values (`true` or `false`). Most often, these operands are simple Boolean expressions but could be a Boolean literal or a non-void method invocation that returns a Boolean value.

Table 4.2 gives the three Java logical operators normally used in control-of-flow statements and the symbols used to represent them. The three operators are shown in decreasing precedence order: the NOT operator has the highest precedence, followed by the `AND` operator, and the OR operator has the lowest precedence. As shown in Appendix E, the arithmetic operators and the relational and equality operators have higher precedence than the logic operators. The Java symbols for the AND (`&&`) and OR (`||`) operators given in the second column of the table are each two keystrokes. The keystrokes (||) used in the symbol for the OR operator is located above the Enter key on the keyboard.

Table 4.2
Java's Logical Operators

Logical Operator	Java Symbol	Examples that Evaluate to true
`Not`	!	! ('p' > 'x')
`And`	&&	8 < 10 && 6 == 2 * 3
`Or`	||	7 < 4 || 8 >= 5

All of the compound logical expressions shown in the rightmost column of Table 4.2 evaluate to `true`. To evaluate the truth value of a complex Boolean expression, we must know the meaning of the conditional logic operators. As previously stated, the meaning of the unary `NOT`(!) operator is simply to reverse the truth value of its operand. For example, because `'p'` comes before `'x'` in the extended ASCII table, (`'p' > 'x'`) evaluates to `false` and !(`'p' > 'x'`) evaluates to `true`. Similarly, `!`(`6 > 10`) evaluates to `true`.

The meaning of the two binary logical operators, AND and OR, is usually conveyed in truth tables such as the one shown in Table 4.3. The four possible combinations of the truth values of their two Boolean operands, A and B, are given in the two columns on the left side of the table. The corresponding values of the AND and OR operators for each of the four possible values of their operands is given in the two columns on the right side of the table. Summarizing the resulting values, `A && B` evaluates to `true` only when `A` and `B` are both `true`, and `A || B` evaluates to `false` only when `A` and `B` are both `false`. The compound Boolean expression in the third row of Table 4.2 evaluates to `true` because one of the operands, 8 >= 5, is `true`.

Table 4.3
Meaning of Java's Binary Logical Operators

Operand Truth Values		Meaning of Operators	
A	**B**	**A && B**	**A || B**
`true`	`true`	`true`	`true`
`true`	`false`	`false`	`true`
`false`	`true`	`false`	`true`
`false`	`false`	`false`	`false`

Figure 4.1 presents an application that evaluates simple Boolean expressions whose operands are literals, primitive and reference variables, and a compound Boolean expression. The output produced by the program is given at the bottom of the figure.

When a compound expression is evaluated, it follows the order of precedence from left to right. Parentheses can also be used to make the ordering clear or to enforce a certain ordering in the evaluation. For example, the expression on line 23 might be written as

```
((i1 == 5 || c1 < 'A') && (d1 != 21.8 ))
```

to specify the order of evaluation. Evaluating the sub-expressions in a different order might give a different result.

```
1    public class BooleanExpressions
2    {
3       public static void main(String[] args)
4       {
5       int i1 = 5;
6       double d1 = 3.53;   double d2 = 54.88;
7       char c1 = 'A';   char c2 ='C';
8       boolean b1 = true;   boolean b2 = false;
9       String s1 = new String("Bob");
10      String s2 = new String("Bob");
11
12      System.out.println(i1 < 5);
13      System.out.println(d1 > d2);
14
15      System.out.println(i1 >= d1); // integer i1 promoted
16      System.out.println(d1 <= 3); // integer 3 is promoted
17
18      System.out.println(c1 < c2); // lexicographical order used
19      System.out.println(10 > c2); // c2 promoted to numeric 67
20
21      System.out.println(b1 == b2) ;
22
23      System.out.println( i1 == 5 || c1 < 'A' && d1 != 21.8 );
24
25      System.out.println(s1 == s2); // compares contents of s1 and s2
26      }
27   }

Output
false
false
true
false
true
false
false
true
false
```

Figure 4.1
The application **BooleanExpressions** and the output it produces.

Lines 5–8 declare and initialize integer, double, character, and Boolean variables. These variables are used in simple Boolean expressions that are evaluated and output on lines 15–21. The types of the operands in the expressions on lines 15, 16, and 19 do not match, so promotion is used before these expressions are evaluated. The contents of the character variables on line 18 are interpreted as numerics before the Boolean expression is evaluated. Because 65 ('A') is less than 67 ('C'), the fifth output is true.

Line 23 contains an example of a compound Boolean expression containing two conditional logical operators OR and AND. Although the order of the operations in this expression is not important, the AND operation, having higher precedence, is evaluated first. This reduces the expression to

```
i1 == 5 || false
```

which evaluates to `true` (the next to the last output in Figure 4.1).

The operands in the Boolean expression output on line 25 are the two string variables declared on lines 9 and 10. Both strings are initialized to "Bob" by the `String` class's one-parameter constructor, yet the comparison for equality on line 25 produces an output of `false` (the last output). This is because the equality operators always compare the contents of the reference variables rather than the contents of the objects they refer to. Because the objects `s1` and `s2` are stored in different locations, the contents of `s1` and `s2` are not equal, and the Boolean expression on line 25 evaluates to `false`. Most often, to compare the contents of two objects, we have to add a method to the object's class that performs the comparison and then returns a Boolean value.

4.2.3 Comparing String Objects

In Chapter 7, we will discuss techniques for comparing the contents of any two objects. Strings are used so often in programs that the `String` class provides several methods for comparing them. One of these is the `equals` method. It is a non-static method that returns a Boolean value. The returned value is `true` when the contents of the string object sent to it is the same as the contents of the string object that invoked it. The comparison of the two strings is case sensitive. The following code fragment demonstrates the use of the method. The first three invocations to the method return `true`, and the last two return `false`.

```
String name1 = new String("Bob");
String name2 = new String("Bob");
String name3 = "BOB";
String name4 = "Mary";
System.out.println(name1.equals(name2));
System.out.println(name1.equals("Bob"));
System.out.println(name1.equals("Bob") || name1.equals("Mary");
System.out.println(name1.equals(name3)); // false, case mismatch
System.out.println(name1.equals(name4)); // false, different names
```

The third invocation demonstrates that a method that returns a Boolean value can be used as an operand in a compound Boolean expression.

The `String` class contains three other non-static methods for comparing strings. Their names are: `equalsIgnoreCase`, `compareTo`, and the `compareToIgnoreCase`. Like the `equals` method, the `equalsIgnoreCase` method returns a Boolean value, which is `true` when the contents of the string object sent to it is the same as the contents of the string object that invoked it. Unlike the `equals` method, case sensitivity is ignored when making the comparison.

The `String` class's `compareTo` and `compareToIgnoreCase` methods determine the relative lexicographical order of two `String` objects. These non-static methods return an integer whose value reflects the lexicographical order of the string that invoked it relative to the string sent to it as an argument. The `compareTo` method considers case sensitivity, and the `compareToIgnore-Case` ignores case sensitivity. The code fragment below compares the lexicographical order two strings `s1` and `s2`:

```
int order1 = s1.compareTo(s2);
int order2 = s2.compareToIgnoreCase(s2);
```

The values returned to the variables `order1` and `order2` would be:

- negative when `s1` comes before `s2` in lexicographical order
- positive when `s1` comes after `s2` in lexicographical order
- zero when `s1` and `s2` are equal in lexicographical order

Although the `compareTo` and the `CompareToIgnoreCase` methods can be used to determine when two strings are equal, it is good coding practice to use the `equals` and `equalsIgnoreCase` methods when testing two strings for equality because it makes our code more readable.

4.3 THE IF STATEMENT

The `if` statement is one of two Java control-of-flow statements that can be used to alter the default sequential execution of a program based on the truth value of a Boolean expression. The other statement is the `if-else` statement, which will be discussed in Section 4.4.

The syntax of the `if` statement begins with the keyword if, followed by a Boolean expression inside parentheses, followed by a statement or group of statements to be skipped or executed. When there is a group of statements, they must be coded as a statement block; that is, they must be enclosed in braces. The group of statements will be executed when the Boolean condition is `true`, and skipped when the Boolean expression is `false`. Thus, the syntax of the statement is:

One Statement Syntax

```
if(a Boolean expression)
a Statement to be skipped
or executed
```

Multiple Statement Syntax

```
if(a Boolean expression)
{ Statement1 to be skipped or executed
                  :
  Statementn to be skipped or executed
}
```

Even when there is just one statement, it is better coding practice to enclose the one statement in braces, which makes the statement more readable and less prone to errors. For example, if during the development of the program we were to decide to add a second statement and neglected to add the open and close braces around the two statements, the second statement would not be considered part of the `if` statement. It would always execute. The two coding examples given below are not equivalent:

```
if(a false Boolean expression)
   statement1
   statement2
```

```
if(a false Boolean expression)
{  statement1
   statement2
}
```

The code on the left always executes `statement2`, even though the indentation seems to imply that its execution is dependent on the truth value of the Boolean expression. A good habit to acquire when writing an `if` statement is to write this code fragment first:

```
if( )
{

}
```

and then fill in the Boolean condition and the statements to be skipped when the condition is `false`. Most often, we see the `if` statement coded as:

```
if(a Boolean expression)
{
  //One or more statements possibly to be skipped
}
```

The meaning, or *semantics*, and the execution path of the `if` statement is illustrated in Figure 4.2

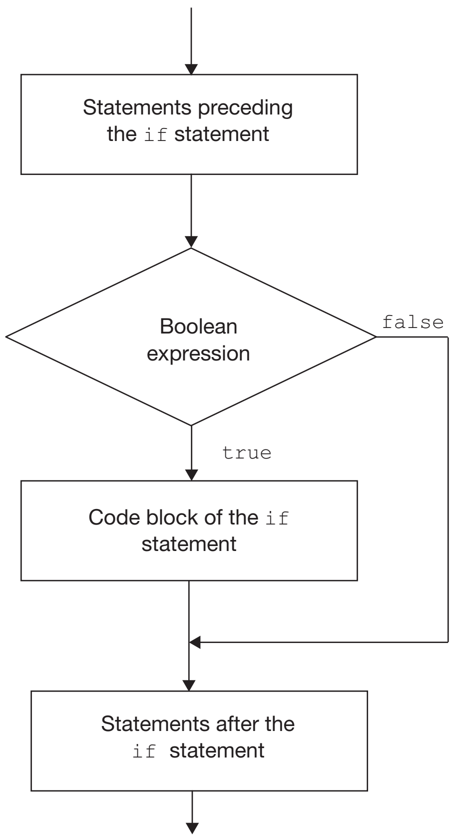


Figure 4.2
The meaning and execution path of the if statement.

We conclude this section with a discussion of a use of the `if` statement, making game objects disappear, and then present a game programming application that demonstrates several uses of the `if` statement.

Using the if Statement

In many games, the game's objects disappear based on events that occur as the game progresses. When Pac-Man collides with a food pellet, the pellet disappears, or when Frogger is hit by a car, she disappears. Often, after the event occurs the object not only becomes invisible, but it is eliminated from the game. Graphic objects can be made to disappear by either drawing them in the color of the program's window (or in our case the game board), or by not drawing them at all.

To convey the visibility status of a game piece object, e.g., a food pellet, a Boolean data member is added to the object's class. When an event occurs that changes the status (for example, when food pellet `p1` is eaten by Pac-Man), the truth value of the data member is reversed by the code that detected the event. The `draw` call back method can use the truth value of this data member in an `if` statement's Boolean condition to decide whether or not to draw the object. If a Boolean data member named `eaten`, initialized to `false`, was added to the class of Pac-Man's pellets, and the data member was set to `true` when the pellet was eaten, then adding the following code fragment to the `draw` call back method would make pellet `p1` disappear after the pellet was is eaten.

```
if( p1.getEaten( ) == false )
{
  p1.show(g);
}
```

If the variable `count` was being used to keep track of the game's time, and pellet `p1` was only to appear after the game had been played for 20 seconds, then a compound Boolean expression would be used in the above code fragment.

```
if( p1.getEaten( ) == false && count >= 20)
{
  p1.show(g);
}
```

In this case the pellet, `p1`, would appear 20 seconds into the game, and it would disappear when an event changes that pellet's data member `eaten` to `true`.

In Section 2.9.1 (Figure 2.12), the counting algorithm was used to keep track of a game's time. Figure 4.3 presents the code discussed in that section with three `if` statements added to it: two to the `draw` call back method (lines 17–31) and one to the `timer1` method (lines 33–40). In addition, a `BoxedSnowman` object `s1`, whose class is given Figure 4.4, has been added to the application (line 10). The graphical output produced by the program is given in Figure 4.5.

When the application shown in Figure 4.3 is launched, the number of elapsed seconds is displayed on the game board starting from zero (top left side of Figure 4.5). To begin the game the start button is clicked, which causes the elapsed time to be updated every second. Five seconds into the game, the snowman `s1` appears at the center of the game board (top right side of Figure 4.5). After ten seconds, the game ends. The elapsed time remains at ten seconds, a message appears on the game board indicating that the game is over, and the snowman disappears from the game board (bottom portion of Figure 4.5).

Line 20 of the application displays the number of elapsed seconds, which is stored in the class variable `count`. This variable is incremented on line 35 of the `timer1` call back method, which (by default) executes once a second. Ten seconds into the game, the Boolean condition of the `if` statement that begins on line 36 becomes `true`, and line 38 invokes the game environment's `stopTimer` method to stop timer 1 from ticking. As described in Appendix B, this method is passed one argument, which specifies the timer number (1, 2, or 3) that is to be stopped. It is a nonstatic method, invoked on the program's `GameBoard` object `gb`, which was declared on line 8.

The Boolean data member `visible` has been declared on line 9 of the `BoxedSnowman` class (Figure 4.4) to store the visibility status of a snowman, and the class contains a `set` and `get` method (lines 51–59) to allow client code to access this private data member. To make the snowman appear after five seconds has elapsed, snowman `s1`'s visibility status is fetched by a call to the `getVisible` method on line 21 of the application, and the returned value is used in the compound Boolean expression to decide when to show the snowman on the game board. The snowman will be shown when its `visible` data member is `true` *and* the game's time is five seconds or greater. Since `visible` is initialized to on line 9 of the `BoxedSnowman` class to `true`, the snowman is displayed on the game board five seconds into the game.

To make the snowman disappear after ten seconds, the `if` statement inside the `timer1` call back method (lines 33–41) sets snowman `s1's` visible property to `false` (line 36) when `count` equals ten. This causes the first term in the Boolean expression on line 21 to become `false`, and line 23, which displays the snowman on the game board, does not execute.

The `if` statement that begins on line 26 displays the game ending messages when the game time reaches ten seconds.

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.Graphics;
3    import java.awt.Font;
4
5    public class IfStatement extends DrawableAdapter
6    {
7      static IfStatement ga = new IfStatement();
8      static GameBoard gb = new GameBoard(ga, "The if Statement");
9      static int count = 0;
10     static BoxedSnowman s1 = new BoxedSnowman(250, 215, Color.BLACK);
11
12     public static void main(String[] args)
13     {
14       showGameBoard(gb);
15     }
16
17     public void draw(Graphics g) // the draw call back method
18     {
19        g.setFont(new Font("Arial", Font.BOLD, 18));
```

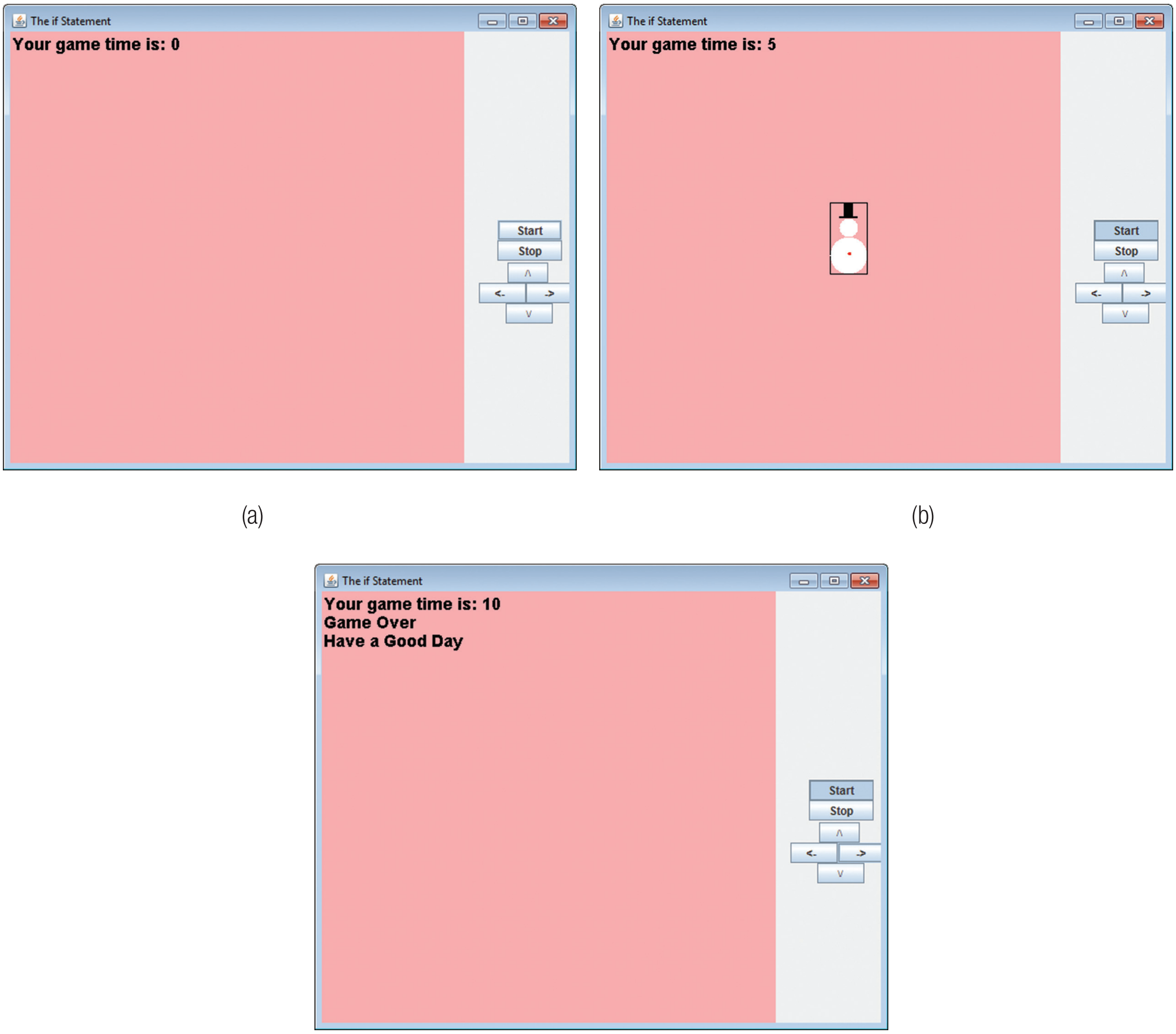
```
        g.drawString("Your game time is: " + count, 10, 50);
        if(s1.getVisible() == true && count >= 5)
        {
           s1.show(g);
        }

        if(count == 10)
        {
           g.drawString("Game Over", 10, 70);
           g.drawString("Have a Good Day", 10, 90);
        }
     }

      public void timer1()
      {
         count = count + 1;
         if(count == 10)
         {
            gb.stopTimer(1);
            s1.setVisible(false);
         }
      }
     }
```

Figure 4.3
The application **`IfStatement`**.

```
import java.awtGraphics;
import java.awt.Color;

public class BoxedSnowman
{
  private int x = 8;
  private int y = 30;
  private Color hatColor = Color.BLACK;
  private boolean visible = true;

  public BoxedSnowman(int intialX, int intialY, Color hatColor)
  { x = intialX;
    y = intialY;
    this.hatColor = hatColor;
  }

  public void show(Graphics g) //g is the game board object
  {
    g.setColor(hatColor);
    g.fillRect(x + 15, y, 10, 15); //hat
    g.fillRect(x + 10, y + 15, 20, 2); //brim
```

```
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y + 17, 20, 20); // head
    g.fillOval(x, y + 37, 40, 40); //body
    g.setColor(Color.RED);
    g.fillOval(x + 19, y + 53, 4, 4); //button
    g.setColor(Color.BLACK);
    g.drawRect(x, y, 40, 77); //inscribing rectangle
  }

  public int getX()
  {
    return x;
  }

  public void setX(int newX)
  {
    x = newX;
  }

  public int getY()
  {
    return y;
  }

  public void setY(int newY)
  {
    y = newY;
  }

  public boolean getVisible()
  {
    return visible;
  }

  public void setVisible(boolean newVisible)
  {
    visible = newVisible;
  }
}
```

Figure 4.4
The **BoxedSnowman** class.



(a) (b)

(c)

Figure 4.5
The output produced by the application **`IfStatement`**.

4.4 THE IF-ELSE STATEMENT

Like the `if` statement, the `if-else` statement is a Java control-of-flow statement that can be used to alter the default sequential execution path of a program by skipping statements based on the truth value of a Boolean expression. This statement can be thought of as having two clauses: an `if` clause and an `else` clause. Each clause has a statement block associated with it. One, and only one, of these blocks will execute. When the Boolean condition is `true`, the statement block associated with the `if` clause executes. When it is `false`, the statement block associated with the `else` clause executes. The syntax of the `if-else` statement is:

```
if(a Boolean expression)
{
   // One or more if clause statements
}
else
{
   // One or more else clause statements
}
```

and its meaning and execution path is given in Figure 4.6

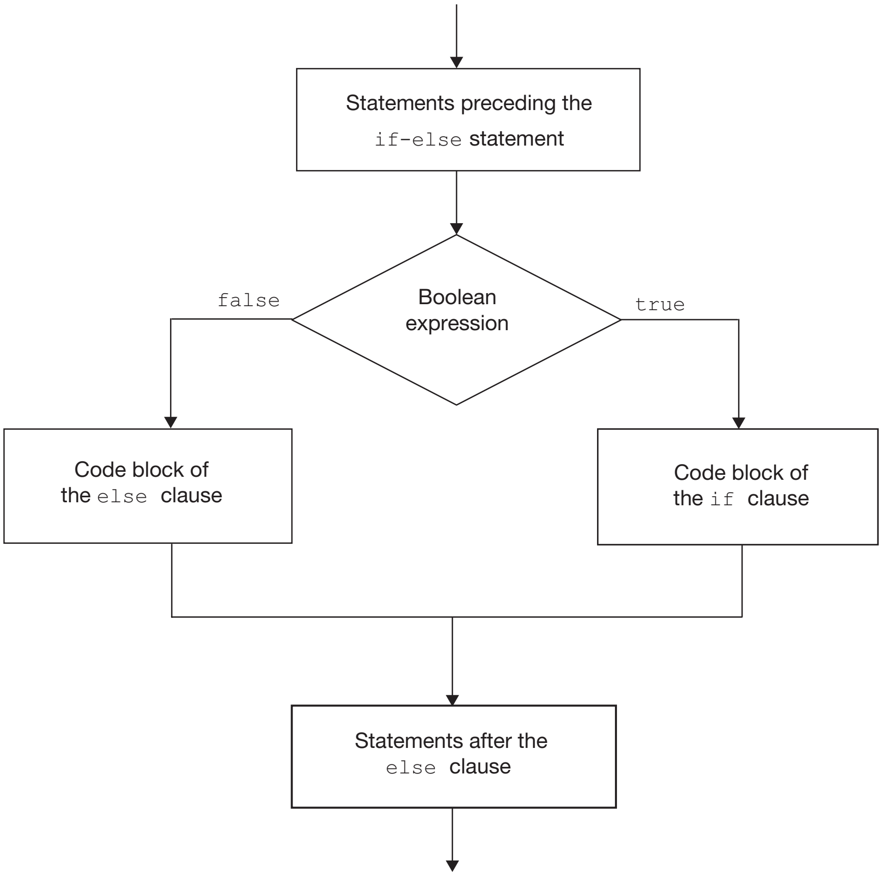


Figure 4.6
The meaning and execution path of the **if-else** statement.

Because the statements in the code block that follow the `else` clause are executed when the `if` statement's Boolean expression is `false`, the `else` clause does not contain its own Boolean expression. The following code fragment determines what weight jacket to wear based on the temperature stored in the memory cell `temperature`:

```
if(temperature <= 45)
{  System.out.println("It is a frigid " + temperature + " degrees,");
```

```
    System.out.println("Wear your heavy jacket.");

}
else
{ System.out.println("It is rather mild " + temperature + " degrees,");
  System.out.println("Wear your light weight jacket");


}
```

The if-else statement is used to choose one of two statement blocks to execute: the first when the if statement's Boolean condition is true and the second when it is false. By coding just one statement into the else clause's statement block that is another if-else statement, we can choose between one of three mutually exclusive alternatives, as illustrated in the following coding template:

```
     if(Boolean expression 1)
{
   // One or more if clause statements in code block 1
}
     else if(Boolean expression 2)
{
   // One or more if clause statements in code block 2
}
     else
{
   // One or more else clause statements in code block 3
}
```

As indicated by the second comment in the template, the second set of open and close parentheses defines the code block of the second if statement. Because the second if statement is the only statement in the first else clause's code block, not coding it inside a set of brackets improves readability. Figure 4.7 illustrates the meaning and execution path of the code template.

This coding process can be progressively repeated when there are more than three mutually exclusive alternatives. The following code template illustrates the use of this concept to choose one of four mutually exclusive code blocks to execute:

```
if(Boolean expression 1)
{
   // One or more if clause statements in code block 1
}
else if(Boolean expression 2)
{
   // One or more if clause statements in code block 2
}
else if(Boolean expression 3)
{
   // One or more if clause statements in code block 3
}
else
```

```
{
   // One or more else clause statements in code block 4
}
```

To improve the readability, it is good programming practice to indent as shown above and to keep the first line of the `if` statements on the same line as the `else` clauses that proceeded them.

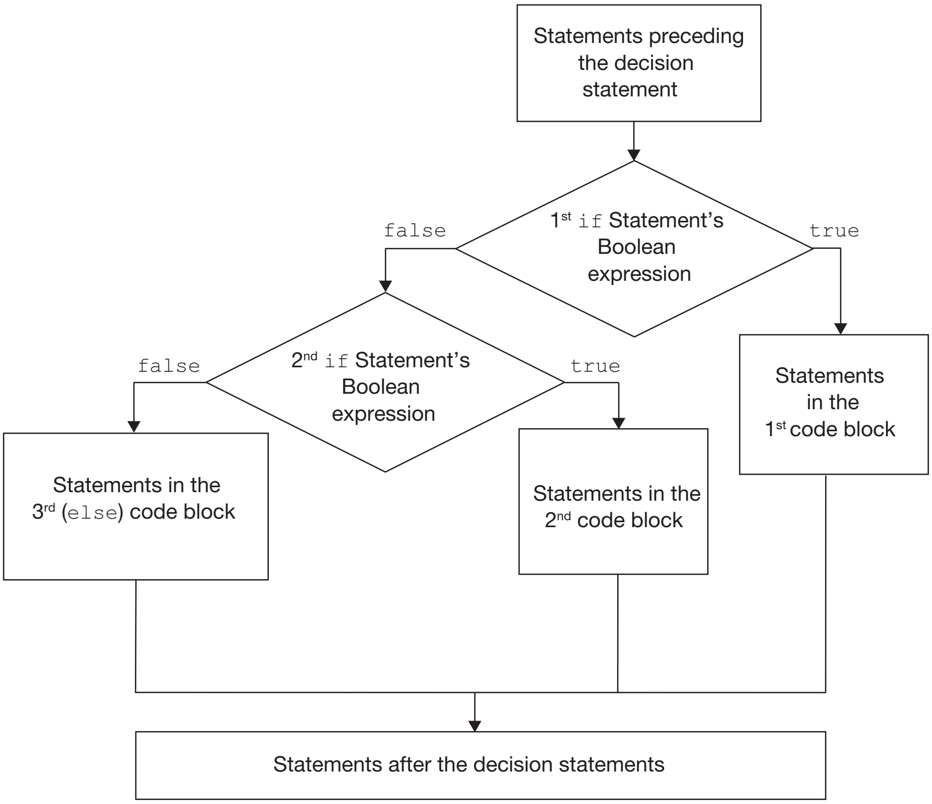


Figure 4.7
The meaning and execution path of an **`if-else`** statement whose **`else`** clause statement is an **`if-else`** statement.

As an example, the following code fragment determines which one of four colors, red, green, blue, or white, was contained in the `String` object `carColor`.

```
if(carColor.equals("Red"))
{
  System.out.println("the car color is Red");
}
else if(carColor.equals("Green"))
{
  System.out.println("the car color is Green");
}
else if( carColor.equals("Blue") )
{
  System.out.println("the car color is Blue");
```

```
}
else
{
  System.out.println("the car color is White");
}
```

These decision statements are executed in the sequence shown in Figure 4.7. The Boolean expressions are evaluated in the order in which they are coded. Only one of the statement blocks will execute, which will be the statement block associated with the first `true` Boolean condition. When none of the Boolean conditions are `true`, the statement block associated with the last `else` clause executes. The last `else` clause and its associated statement block are optional. When it is included, one and only one statement block in the construct always executes.

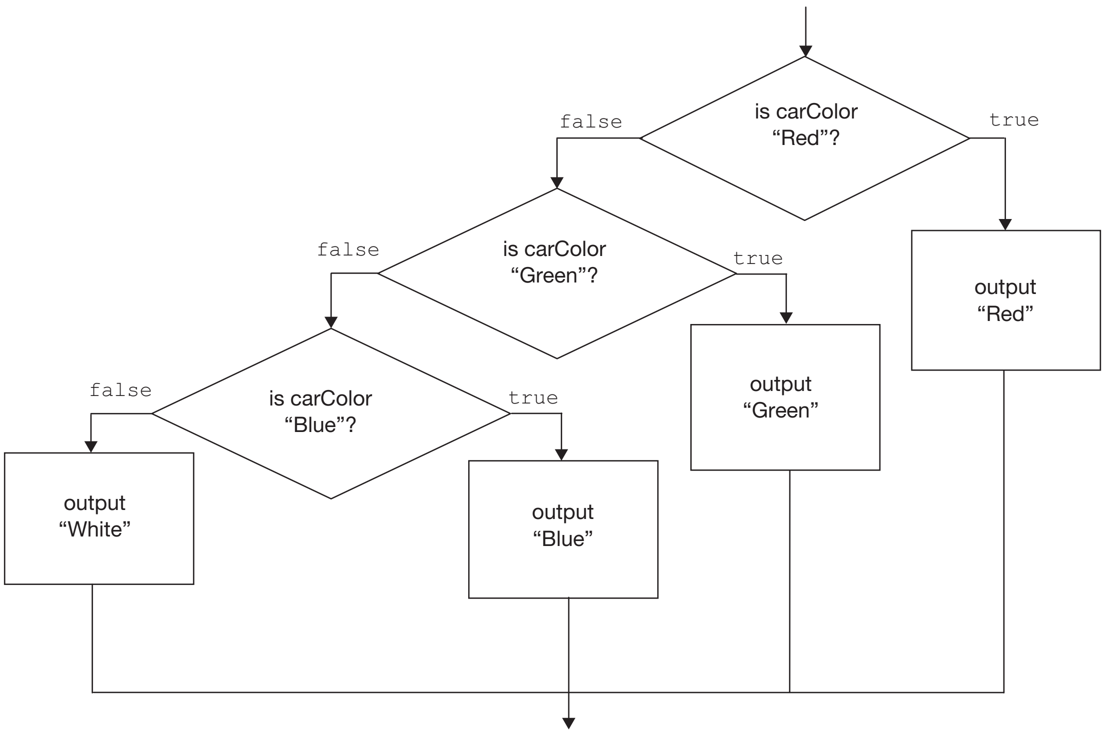


Figure 4.8
Determining the color contain in the **`String`** object **`carColor`**.

We conclude this section with a discussion of a common use of the `if-else` statement (detecting collisions between game pieces) and then present a game-programming application that utilizes this common game event.

Detecting Collisions: Use of the `if` and `else-if` Statements

Most games involve some sort of interaction between the game-piece objects. For example, the ball in a Pong game rebounds off a paddle, the frog in a Frogger game is hit by a truck, or a meteorite collides with a space craft. All of these interactions are referred to as collisions, and usually the score or the length of the game is influenced by these collisions. The Boolean conditions in an `if-else` construct are used to detect the occurrence of collisions, and the code blocks inside the construct are used to take the appropriate action (e.g., change the score or end the game) when a collision occurs.

There are several algorithms used to detect collisions, all of which involve the use of decisions statements. In one of the simplest algorithms, we imagine a rectangle enclosing each game piece. That is, the entire game piece is inscribed inside a rectangle, as shown in Figure 4.9, and the location of the upper-left corner of the rectangle is the game piece's (x, y) location. Then, we consider two objects to be in a collided state when their rectangles touch or overlap.

For example, consider the two snowmen `s1` and `s2` depicted in Figure 4.9a that are 40 pixels wide and 77 pixels high. If snowman `s2` were moving to the left, a collision with snowman `s1` would occur when the left side of `s2`'s rectangle was at the same x location as the right side of `s1`'s rectangle. This situation is depicted in Figure 4.9b. The following Boolean expression, which is true when this event occurs, can be used to detect this collision state.

```
s2.getX( ) == s1.getX + 40; // The snowmen are 40 pixels wide
```

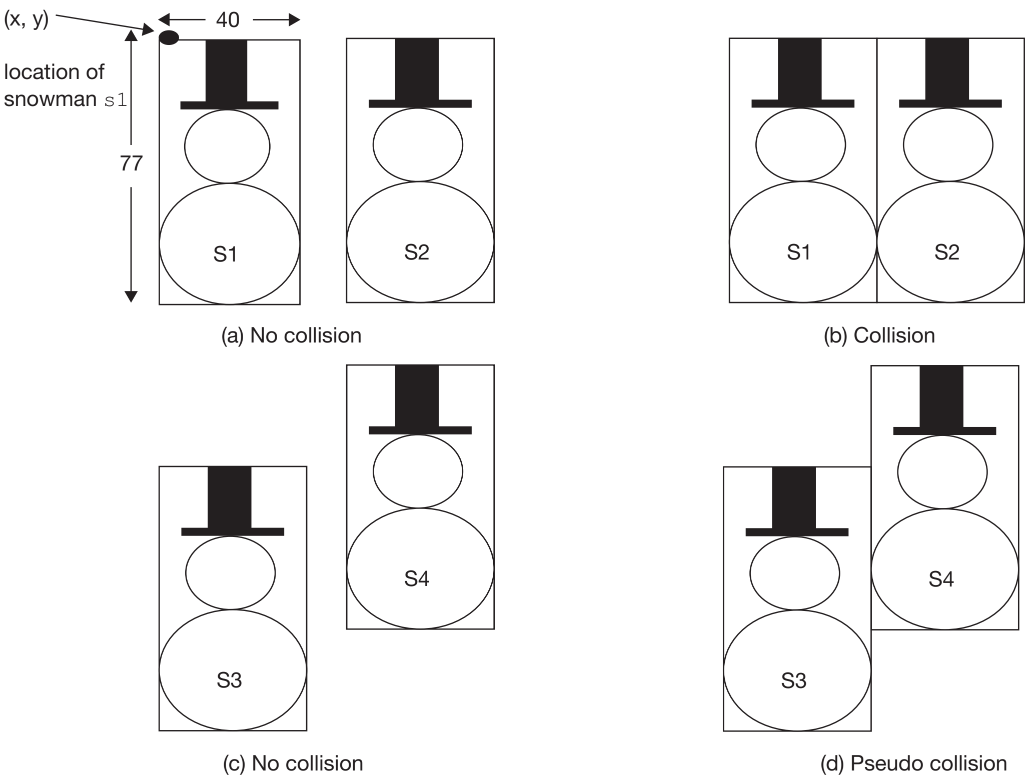


Figure 4.9
Noncollided and collided game pieces.

Although this collision detection scheme is simple, it is not always accurate. When the rectangles of the two snowmen depicted in Figure 4.9b are at the same x location, the bodies of the two snowmen are touching each other. This is not the case for the two snowmen, `s3` and `s4`, shown in Figure 4.9c. If snowman `s4` were moving to the left when the left side of its rectangle is at the same x location as the right side of `s3`'s rectangle, as shown in Figure 4.9d, the two snowmen would not be in a collided state. There would still be a small amount of separation between the left side of `s4`'s body and the right side of `s3`'s head.

Fortunately, in most cases, the game's player would not notice the separation and would visually confirm this pseudo-collision as an actual collision. If we are willing to accept this limitation of our collision-detection scheme, we can extend this simple scheme to detect a collision between the two snowmen as they approach each other from any direction.

Figure 4.10 depicts snowman `s2` in the following four positions relative to snowman `s1`:

- Position 1: `s2` is to the right of `s1`
- Position 2: `s2` is to the left of `s1`
- Position 3: `s2` is below `s1`
- Position 4: `s2` is above `s1`

When snowman `s2` is in any of these positions relative to snowman `s1`, then the two snowmen cannot be in a collided state. In fact, `s2` could be in two of these positions simultaneously, e.g., to the right and above of snowman `s1`, which would also be a non-collided state.

Each of the four positions depicted in Figure 4.10 can be easily detected with a simple Boolean expression. Assuming the snowman is inscribed inside a rectangle that is w pixels wide and h pixels high, the right column of Table 4.4 gives the Boolean conditions that evaluate to `true` when the snowmen are in each of the four positions.

Table 4.4
Boolean Expressions to Detect the Four Non-collided Positions in Figure 4.8

Position of Snowman `s2` Relative to `s1`	Boolean Expression to Detect the Position
1. `s2` is to the right of `s1`	`s2.getX( ) > s1.getX( ) + w`
2. `s2` is to the left of `s1`	`s2.getX( ) + w < s1.getX( )`

These four Boolean expressions can be used in `if-else` statements to determine when the two snowmen have not collided; otherwise they have collided.

```
if(s2.getX() > s1.getX() + w) // s2 right of s1
{
  System.out.println("no collision");
}
else if(s2.getX() + w < s1.getX()) // s2 left of s1
{
  System.out.println("no collision");
}
else if(s2.getY() > s1.getY() + h) // s2 below s1
```

```
{
  System.out.println("no collision");
}
else if(s2.getY( ) + h < s1.getY( )) // s2 above s1
{
  System.out.println("no collision");
}
else // collision
{
   System.out.println("collision");
}
```

Alternately, the Boolean conditions could be combined to form a compound Boolean condition that would evaluate to `true` for a non-collision.

```
(s2.getX( ) > s1.getX() + w || s2.getX() + w < s1.getX() ||
 s2.getY( ) > s1.getY() + h || s2.getY() + h < s1.getY())
```

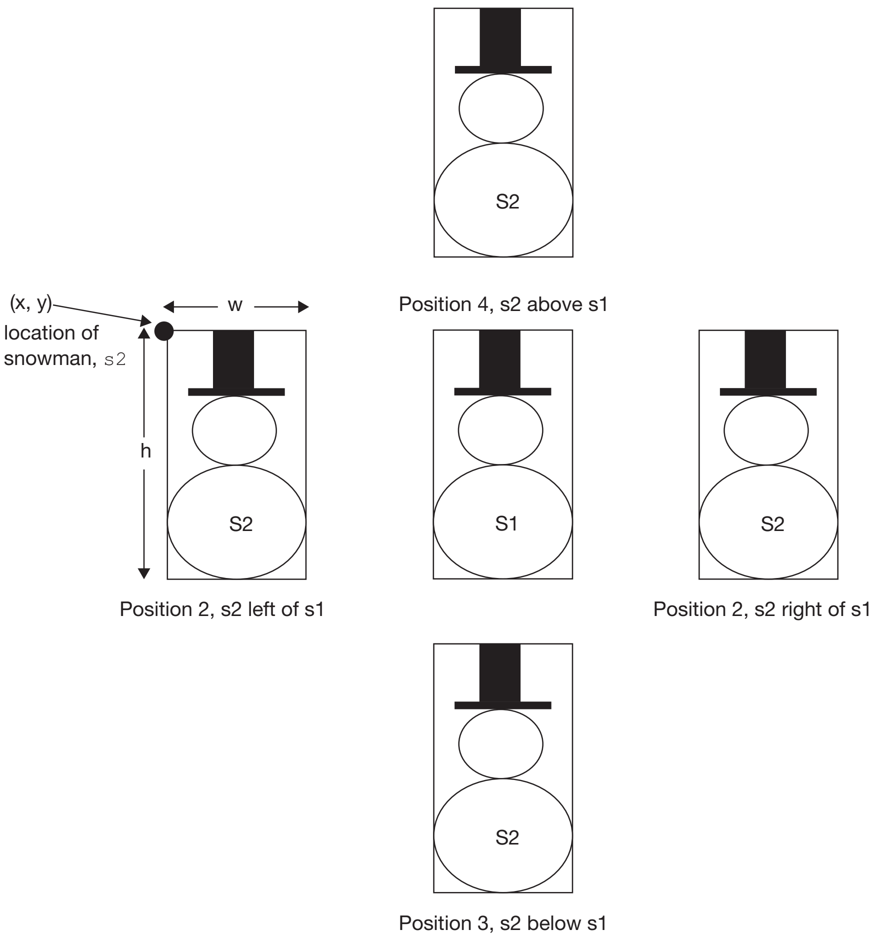


Figure 4.10
The non-collided positions of snowman **s2** relative to snowman **s1**.

Using this compound Boolean expression, the series of `if-else` statements to detect a collision would become

```
if(s2.getX( ) > s1.getX() + w || s2.getX() + w < s1.getX() ||
   s2.getY( ) > s1.getY() + h || s2.getY() + h < s1.getY())
{
  System.out.println("no collision");
}
    else //collision
{
  System.out.println("collision");
}
```

The truth value of the Boolean condition could also be reversed, using Java's `not` (!) logical operator, and the `if` clause of the `if-else` statement would detect a collision between the two snowmen.

```
if( !(s2.getX( ) > s1.getX() + w || s2.getX() + w < s1.getX() ||
  s2.getY( ) > s1.getY() + h || s2.getY() + h < s1.getY()) )
{
  System.out.println("collision");
}
else // no collision
{
  System.out.println("no collision");
}
```

The following code fragment uses an expanded version this `if-else` statement's Boolean expression to detect when a collision occurs and snowman `s2` is in a visible state. When this occurs, the game's score (the variable `score`) is increased by 1, and snowman `s1`'s visible property is set to `false`.

```
if( !(s2.getX( ) > s1.getX() + w || s2.getX() + w < s1.getX()||
      s2.getY( ) > s1.getY() + h || s2.getY() + h < s1.getY()) &&
      s1.getVisible == true) // collision and s1 is visible
{
    score = score + 1;
    s1.setVisible(false);
}
```

An additional term has been added at the end of the Boolean expression. Because it is preceded by the `&&` (`AND`) operator, the expanded expression is only `true` when the two snowmen collide *and* snowman `s1` is visible. This prevents the score from increasing when a game object (i.e., `s2`) collides with an invisible game object that is no longer part of the game (i.e., `s1`).

4.4.1 Conditional Expressions

A conditional expression is a succinct alternative way of a writing a simple `if-else` statement, whose code blocks each contain one statement, that produces a value usually of the same type. For example they both perform an integer calculation or invoke the same non-void method. The conditional expression contains the two characters of the conditional operator, `?:`, and three operands.

The first operand must evaluate to a Boolean condition. This is followed by the question mark of the conditional operator, which is followed by the two operands separated by the conditional operator's colon. The first of these two operands becomes the value of the conditional expression when the Boolean condition is `true`, otherwise the second operand is the value of the conditional expression when the condition is `false`. For example, the conditional expression (a < b ? a * 2 : b * 2) evaluates to a * 2 when a is less than b otherwise it evaluates to b * 2. The value produced by the conditional expression must be used within a line of Java code: e.g., assigned to a variable, or output to a file or to the system console as shown below.

```
1 System.out.println(a < b ? a * 2 : b * 2);
2 int c = a < b ? a * 2 : b * 2;
```

Equivalent to Line 1 above

```
if(a < b)
{
  System.out.println(a * 2);
}
else
{
  System.out.println(b * 2);
}
```

Equivalent to Line 2 above

```
if(a < b)
{
  c = a * 2;
}
else
{
  c = b * 2;
}
```

When the types of the values produced by the code blocks are different, the values must be able to be promoted, or cast into the same type. For example, if one value is a double and the other is an integer the integer would be promoted to a double, unless the double was cast to an int. If the values are addresses of objects, they must be instances of the same class or, we must utilize the techniques of inheritance and polymorphism discussed in Chapter 8 to reconcile the differences.

4.5 NESTED IF STATEMENTS

Just as the `else` clause of an `if-else` statement can contain an `if` statement, the statement block of an `if` statement can also contain other `if` statements. This method of coding is referred to as nested `if` statements, because the second `if` statement can be thought of as an egg inside the nest formed by the first `if` statement's code block.

The following code fragment contains a Boolean variable `raining` and an integer variable `temperature`, and uses a nested `if` statement to determine if a sweater and a raincoat should be carried on a cold day when it is raining.

```
if(raining == true)
{
  System.out.println("Take your umbrella, ");
  if(temperature <= 50) // begins a nested if-else statement
  {
     System.out.println("take a sweater, ");"
     System.out.println("and your raincoat.");
  }
}
```

An `if-else` statement can also be nested inside an `if` statement as demonstrated in the below code fragment:

```
if(raining == true)
{
  System.out.println("Take your umbrella, ");
  if(temperature <= 50) // begins a nested if statement
  {
     System.out.println("take a sweater, ");"
     System.out.println("and your raincoat.");
  }
  else // temperature is > 50 degrees
  {
     System.out.println("and your raincoat");
  }
}
```

The `else` clause in an `if` statement is always paired with the `if` statement whose code block ends just before the `else` clause. The indentation used in the code fragment above is considered good programming practice because it implies this pairing: the `else` clause is part of the `if` statement that checks the temperature. This code fragment is equivalent to the code fragment below, which is considered to be poor programming style because its indentation erroneously implies that the `else` clause is part of the `if` statement that determines if it is raining.

```
if(raining == true)
{
  System.out.println("Take your umbrella, ");
  if(temperature < 50) // begins a nested if statement
  {
     System.out.println("and carry your raincoat too");
  } // end of the inner if statement
else
{
     System.out.println("but not your raincoat");
}
{ // end of the outer if statement
```

The following code segment is another example of the use of a nested `if` statement. It is an alternate way of determining when snowmen `s1` and `s2` have collided *and* `s1` is visible.

```
if( !(s2.getX( ) > s1.getX() + w || s2.getX() + w < s1.getX()||
      s2.getY( ) > s1.getY() + h || s2.getY() + h < s1.getY()))//collision
{
      if(s1.getVisible == true) // and s1 is visible
      {
        score = score + 1;
        s1.setVisible(false);
      }
}
```

4.6 THE SWITCH STATEMENT

The `switch` statement is another control-of-flow statement available in Java. It is not as versatile as the `if` and `if-else` statements in that the decisions these statements make cannot be based on an explicitly written simple or compound Boolean expression. The syntax of the `switch` statement limits the operator used in its decision making to equality. In addition, the equality must be between:

- two `String` objects
- two `byte`, `short`, `char`, or `int` primitive-data types (or classes that "wrap" these data types), or
- two instances of a previously defined enumerated type (which will be discussed in Chapter 7)

All uses of the `switch` statement can be coded using an `if-else` statement, but not vice versa. That being said, there are times when the use of the `switch` statement makes our programs more readable and therefore easier to understand, modify, and maintain. It can only be used when the decision as to which statements to execute and which statements to skip is based on a choice selected from a group, or menu, of *finite* choices. When this is the case, the use of the `switch` statement is considered to be good programming practice.

The syntax of the `switch` statement is depicted in Figure 4.11. The indentation used in the figure also reflects good programming practice.

```
switch (choiceExpression)
{
  case choiceValue1:
  {
        // statement block for choiceValue1
        break;
  }
  case choiceValue2:
  {
        // statement block for choiceValue2
        break;
  }
         :
         :
  case choiceValueN:
  {
        // statement block for choiceValueN
        break;
  }
  default:
  {
        // default statements
  }
}
```

Figure 4.11
The syntax of the **`switch`** statement.

As shown in the figure, the first line of the statement begins with the keyword **`switch`**, and the remaining lines of the statement consist of `case` clauses and a `default` clause enclosed in a set of brackets. When typing the statement, it is best to begin by typing the following required syntax and then filling in the remainder of the statement's first line and the `case` and `default` clauses that are appropriate to the particular use of the statement.

```
switch( )
{

}
```

Referring to Figure 4.11, the three most common (and difficult to discover) syntax errors made when coding a `switch` statement are:

1. neglecting to code the open and close parentheses after the keyword `switch`
2. coding a semicolon after the close parenthesis on the first line of the statement
3. neglecting to code the colon (not semicolon) after the `choiceValue1`, or `choiceValue2`... or after the keyword `default`

The entity enclosed in the parentheses after the keyword `switch` is referred to as the *choice expression*. The choice expression must be a variable whose type is one of the allowable types previously mentioned (e.g., a `String` object, an integer variable, etc.) or it can be an expression that evaluates to one of these types.

```
switch(choice)
{
   case choiceValue1:
   {
     // statements for choiceValue1
     break;
   }

   case choiceValue2:
   {
     // statements for choiceValue2
     break;
   }
      :
      :
   case choiceValueN:
   {
     // statements for choiceValueN
     break;
   }
   default:
   {
     // default statements
   }
}
```

(a)

```

if (choice == choiceValue1)
{
  // statements for choiceValue1

}

else if(choice == choiceValue2)
{
  // statements for choiceValue2

}
    :
    :
else if(choice == choiceValueN)
{
  // statements for choiceValueN
}

else
{
  // default statements
}
```

(b)

Figure 4.12
Semantically equivalent **`switch`** and **`if-else`** statements.

When a switch statement begins execution, the value of the choice expression is determined and then the statement block of the *first* case clause whose choice value is equal to that value is executed. If the choice expression is not equal to one of the choice values, the default clause's statement block executes. Figure 4.12 illustrates the meaning and execution path of a switch statement (Figure 4.12a) by comparing it with an equivalent if-else statement (Figure 4.12b).

As an example, the following code fragment determines which one of four colors, red, green, blue, or white, is contained in the String object carColor:

```
switch (carColor)
{
  case "red":
  {
     System.out.println("the car color is red");
     break;
  }
  case "green":
  {
     System.out.println("the car color is green");
     break;
  }
  case "blue":
  {
     System.out.println("the car color is blue");
     break;
  }
  default:
  {
     System.out.println("the car color is white");
  }
}
```

The following code fragment illustrates the use of an arithmetic expression as the choice expression in a switch statement:

```
int i;
String s = JOptionPane.showInputDialog("enter an integer");
i = Integer.parseInt(i);

switch (i * 2)
{
  case 10:
  {
     System.out.println("two times the number is 10");
     break;
  }
  case 20:
  {
     System.out.println("two times the number is 20");
     break;
  }
```

```
    default:
    {
       System.out.println("two times the number is not 10 or 20");
    }
}
```

There is no limit to the number of case clauses that can be used in a `switch` statement. The `default` clause is optional and, if used, must be coded as the last clause in the statement. The brackets surrounding the statements in the `case` and `default` clauses are not necessary and are only used to improve readability.

Several cases can be assigned to the same statement block using the syntax

```
case 2: case 5: case 7:
{
  // statement block for all three cases
   break;
}
```

The above statement block would execute when the choice expression evaluates to 2, 5, or 7.

The `break` statement at the end of the code block of each `case` is also optional. However, unlike the optional bracket pairs, its presence has a major impact on the execution path of the construct. A `break` statement is a control-of-flow statement that does not use a logical expression to decide when to execute or skip statements. Rather, when a `break` statement is executed inside a `switch` statement, it always ends the execution of the `switch` statement in which it is coded. Basically, it means: break out of this statement. It can also be used inside `if` or `if-else` statements to end their execution.

When a `break` statement inside a control-of-flow statement is executed, the next statement to execute is the one that immediately follows the control-of-flow statement. When executed inside a `switch` statement, the statement blocks in all of the subsequent `case` clauses and the statement block in the `default` clause are skipped, and the next statement to execute is the one that follows the close brace at the end of the `switch` statement. (That is, the close brace that is paired with the open brace after the keyword `switch`.)

When the `break` statement is not coded at the end of a `case` clause, after the statements in that clause execute, the statements in all subsequent case clauses execute until a `break` statement is encountered. If a `break` statement is not encountered, the `default` clause also executes. Because most times the choices coded into the `switch` construct are mutually exclusive, a `break` statement is usually coded as the last statement in each `case` clause.

Figure 4.13 shows a game application that uses the `switch` and `break` statements to change the position of a snowman on a game board, uses the `if` and `if-else` statements to determine the game's score, make a second snowman disappear and then reappear at a new location, and to determine when the game is over.

When the application is launched, two snowmen, one wearing a black hat and the other wearing a green hat, appear on the game board below the game's score and remaining time (Figure 4.14a). The game begins when the player clicks the Start button on the game board. The objective of the game is to make the two snowmen collide as many times as possible before time runs out, using the keyboard cursor control keys to move the black-hat snowman. Each time they collide, a point is awarded and the green-hat snowman disappears. It reappears at a new location after the black-hat snowman has been moved to a location such that the two snowmen are no longer in a collision state.

The game's snowmen, `s1` and `s2`, are instances of the `BoxedSnowman` class (Figure 4.4). They are created on lines 9 and 10 of the application shown in Figure 4.13 using a three parameter constructor to specify the snowmen's position and hat color: `s1` green, `s2` black. Line 29 of the `draw` call back method outputs the remaining time, and line 60 outputs the player's score just before the draw method ends.

Lines 54–58 invokes the `BoxedSnowman` class's `show` method to draw the snowmen on the game board at their current (x, y) locations. The `if` statement that begins on line 55 checks the visibility status of snowman `s1` to decide if it should be drawn (line 57). The initial value of a `BoxedSnowman`'s visible property is `true` (Figure 4.4, line 9), so when the game is launched, it appears on the game board.

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3    //Use of decision statements
4
5    public class DecisionsControlOfFlow extends DrawableAdapter
6    {
7       static DecisionsControlOfFlow ge = new DecisionsControlOfFlow();
8       static GameBoard gb = new GameBoard(ge, "Control Of Flow");
9       static BoxedSnowman s1 = new BoxedSnowman(300, 200, Color.GREEN);
10      static BoxedSnowman s2 = new BoxedSnowman(30, 100, Color.BLACK);
11      static int score = 0;
12      static int count = 10;
13
14      public static void main(String[] args)
15      {
16         showGameBoard(gb);
17      }
18
19      public void draw(Graphics g) //call back method
20      {
21         int w = 40;
22         int h = 77;
23         int s1X, s1Y, s2X, s2Y, temp;
24
```

```
      s1X = s1.getX(); s1Y = s1.getY();
      s2X = s2.getX(); s2Y = s2.getY();
      g.setColor(Color.BLACK);
      g.setFont(new Font("Arial", Font.BOLD, 18));
      g.drawString("Time remaining: " + count, 260, 50);

      if(count == 0) //game over
      {
        g.setColor(Color.BLACK);
        g.drawString("Game Over", 205, 70);
        g.drawString("Have a Good Day", 175, 90);
      }
      else if( !(s2X > s1X + w || s2X + w < s1X || s2Y > s1Y + h ||
          s2Y + h < s1Y) && s1.getVisible() == true) // collision
      {
        score = score + 1;
        s1.setVisible(false);
      }
      else if( s2X > s1X + w || s2X + w < s1X || s2Y > s1Y + h ||
            s2Y + h < s1Y) // no collision
      {
        if(s1.getVisible() == false) // not visible
        {  temp = s1.getX();
           s1.setX(s1.getY());
           s1.setY(temp);
           s1.setVisible(true);
        }
      }

      s2.show(g);
      if(s1.getVisible() == true)
      {
        s1.show(g);
      }
      g.setColor(Color.BLACK);
      g.drawString("Score: " + score, 150, 50);
    }

    public void keyStruck(char key) // call back method
    {
      int newX, newY;

      switch (key)
      {
        case 'L':
```

```
70            {
71               newX = s2.getX() - 2;
72               s2.setX(newX);
73               break;
74            }
75            case 'R':
76            {
77               newX = s2.getX() + 2;
78               s2.setX(newX);
79               break;
80            }
81            case 'U':
82            {
83               newY = s2.getY() - 2;
84               s2.setY(newY);
85               break;
86            }
87            case 'D':
88            {
89               newY = s2.getY() + 2;
90               s2.setY(newY);
91            }
92        } // end of switch statement
93     }
94     public void timer1() // call back method
95     {
96        count = count - 1;
97        if(count == 0)

98        {
99           gb.stopTimer(1);
100       }
101    }
102 }
```

Figure 4.13
The **DecisionsControlOfFlow** application: A decision statement case study.

The use of a `switch` statement is illustrated on lines 67–92. In this case, the `switch` statement is used to determine which of the four cursor-control keyboard keys was struck to move the snowman `s2` two pixels from its current location. The statement is coded inside the game environment's call back method `keyStruck` (line 63), which is invoked by the game environment every time a keyboard key is struck. The method has one parameter named `key` whose type is `char`, and the game environment passes a character, the key that was struck, into it. After `keyStruck` completes its execution, the game environment invokes the `draw` call back method.

The parameter `key` on line 63 is used as the `switch` statement's choice expression on line 67. When the keyboard left, right, up, or down cursor-control keys are struck, they generate the characters `'L'`, `'R'`, `'U'`, or `'D'`, respectively. These characters are used as the `switch` statement's cases on lines 69, 75, 81, and 87 to decide in which direction to move snowman `s2`.

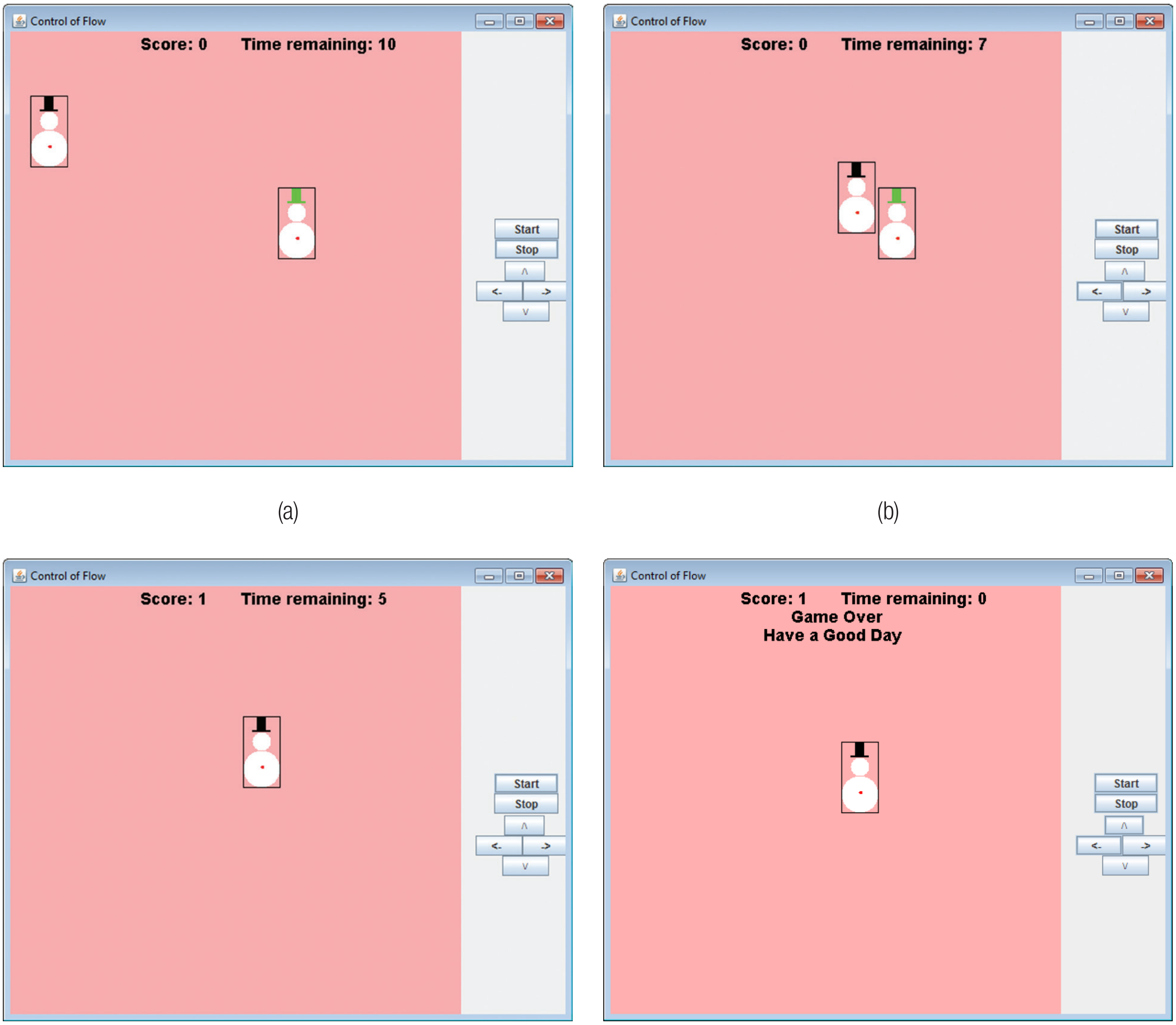
! TIP

When a key on the keyboard is held down, it transmits characters 20 times a second just as if the key was being pressed and released 20 times a second. For this reason, to control the motion of game pieces, key strokes are preferred over button clicks.

Figure 4.14b shows the game board three seconds after the Start button was clicked and the right and down cursor keys were used to move snowman `s2` adjacent to snowman `s1`. One more right cursor keystroke will cause a collision.

Line 31 begins an `if-else` statement that contains a nested `if-else` statement (line 37) and two nested `if` statements (lines 43 and 46). The keyword `else` that appears on lines 37 and 43 are part of the `if-else` statements that begin on lines 31 and 37, respectively. Line 31 decides if the game is over, and when it is, it announces it to the game's player.

The `if-else` statement that begins on line 37 decides if the two snowmen have collided when snowman `s2` is visible. Its Boolean expression, as discussed at the end of Section 4.4, is true when it is not the case that snowman `s2` is to the right, to the left, or below or above snowman `s1`, and `s1` is visible. When this is the case, the `if` clause's code block increases the player's score by one point using the counting algorithm (line 40) and sets the visible property of snowman `s1` to `false` (line 41). Setting `s1`'s `visible` property to `false` causes it to disappear from the game board (Figure 4.14c) because the Boolean condition in the `if` statement that draws `s1` (line 55) is now `false`. The setting of `s1`'s visible property to `false` also prevents the awarding of points until `s1` is again visible which occurs when the two snowmen are no longer in a collision state. The determination that the two snowmen are no longer in a collision state is performed by the `if` statement on line 43. Its Boolean condition is the same as the condition on lines 37 and 38, except that the NOT (`!`) operator and the test for visible have been removed. This Boolean condition is `true` when the snowmen are not in a collision state. Then the nested `if` statement that begins on line 46 executes and decides if snowman `s1` is invisible. When it is invisible, the nested `if` statement's code block executes relocating snowman `s1` by swapping its x and y coordinates. This code block also sets `s1`'s `visible` property to `true` (line 50), which causes the `if` statement that begins on line 55 to draw snowman `s1` on the game board at its new location (Figure 4.14d).



(a) (b) (c) (d)

Figure 4.14
The output of the **`DecisionsControlOfFlow`** application.

4.7 CONSOLE INPUT AND THE SCANNER CLASS

We have already learned how to perform input and output using message dialog boxes and how to send output to the system console using the `println` method. The system console can also be used to perform keyboard input using methods in the `Scanner` class. These nonstatic methods can also be used to perform input from a disk file, which will be discussed in the next section.

Just as there is a predefined output object attached to the system console, `System.out`, there is a predefined input object attached to the console, `System.in`. However, before we use the input methods in the `Scanner` class we have to declare a `Scanner` object and pass the console object to the `Scanner` class's one-parameter constructor. The following code fragment declares the `Scanner` object `consoleIn`:

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

The `Scanner` object, `consoleIn`, can then be used to invoke non-void methods in the `Scanner` class, which accept input from the system console. As the user types the input, the keystrokes are output to the console. The names of the three most frequently used methods in this class all begin with the word next. Their full names and the type of data they return are given in Table 4.5. To use the methods, include the following import statement in your program:

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

Table 4.5
Commonly Used Input Methods in the **`Scanner`** Class

Method Name	Returned Type
`nextInt`	`int`
`nextDouble`	`double`
`nextLine`	`String`

As their names imply, `nextInt` and `nextDouble` are both used to accept numeric input. When the input is an integer, `nextInt` is used, and `nextDouble` is used when the input is a real number. Both methods parse the input characters into a numeric value, and so there is no need to use the parsing methods in the wrapper classes `Integer` and `Double`. The method `nextLine` is used to accept `String` input.

When these methods are invoked, the program's execution is suspended until the user completes the keyboard input by striking the Enter key. Then, the methods return the input value. Until that point, the user can edit the input using the Backspace and Delete keys. It is good programming practice to precede the method invocations with a well-composed prompt output to the system console. The following code fragment accepts a person's name, age, and weight entered into the system console:

```
Scanner consoleIn = new Scanner(System.in);
String name;
int age;
double weight;

System.out.print("Enter your name: ");
name = consoleIn.nextLine();

System.out.print("\nEnter your age: ");
age = consoleIn.nextInt();

System.out.print("\nEnter your weight: ");
weight = consoleIn.nextDouble();
```

Several numeric inputs can be entered on one line as long as they are separated (delimited) by at least one space. Spaces that precede a numeric input are ignored. The following code segment accepts a person's age and weight input on one line to the system console.

```
int age;
double weight;
Scanner consoleIn = new Scanner(System.in);

System.out.print ("Enter your age and weight on one line " +
  "separated by at least one space: ");
age = consoleIn.nextInt();
weight = consoleIn.nextDouble();
```

! TIP Several numeric inputs, separated by at least one space, can be input on the same line.

Spaces that precede a string input are not ignored. They are considered, and become, part of the input string. Spaces typed after a numeric input will become part of a string input subsequently read from the same line. For this reason, strings should not be input on the same line as numeric inputs.

! TIP Numeric and string inputs should not be input on the same line.

When a numeric input and a string are read from two separate input lines, and the numeric input precedes the string, two invocations of `nextLine` are required to capture the string. This is because numeric inputs leave the character generated by the Enter key "behind," and the `nextLine` method considers this a valid input string (the empty string ""). The following statements accept a person's age, followed by the person's name and address. The first string input is properly preceded by an additional invocation of the `nextLine` method.

```
Scanner consoleIn = new Scanner(System.in);
int age;
String name;
String address;

System.out.print("\nEnter your age: ");
age = consoleIn.nextInt();

consoleIn.nextLine(); // clears the enter keystroke left behind
System.out.print("Enter your name: ");
name = consoleIn.nextLine();

System.out.print("Enter your address: ");
address = consoleIn.nextLine();
```

To fully understand `Scanner` class input, we must recognize that the characters the user types are transferred to a memory resident storage area called an input *buffer*. When a `Scanner` method is invoked and the buffer is empty, the method pauses until an Enter key is struck. If the buffer is not empty, the method accepts an input from the buffer, and then the input is deleted from the buffer. The `nextLine` method also deletes the Enter keystroke from the buffer; however, `Scanner` methods that return numeric values do not remove this character from the buffer.

When reading numeric inputs, this is not a problem because the numeric input methods not only skip leading spaces in the buffer, they also skip the Enter keystroke. This whitespace is ignored until the buffer is empty or they find an input to process. However, the `newline` method does not skip the Enter keystroke. As a result, when an invocation to `nextLine` follows a numeric input it encounters a nonempty buffer containing an Enter keystroke. The `nextLine` method reads and removes the Enter keystroke from the buffer and returns the empty string ("").

NOTE

When reading a string from the console after a numeric input, two invocations of the `newLine` *method are required to read the string. The first invocation flushes the new line (empty string) from the buffer.*

Figure 4.15 presents an application that demonstrates the use of the `Scanner` class's methods to accept input from the system console. The console inputs and corresponding outputs are given at the bottom of the figure.

Line 1 imports the `Scanner` class's methods into the application, and line 6 uses the `Scanner` class's one-parameter constructor to create the object `consoleIn` passing it the predefined console input object `System.in`. Lines 11–16 accept a string, integer, and a double from the system console, each input on a separate line. These values are output on lines 17–18.

Lines 20-25 change the order of the inputs beginning with two numeric inputs on the same console line (lines 20–22). Then, a string is input (lines 23–25). Line 24 clears the Enter keystroke left in the buffer after the second numeric is read (line 22). Lines 26–27 outputs the second set of inputs. Referring to the bottom of Figure 4.15, the user entered several spaces between the input age, 5, and the input weight, 35. These spaces were ignored by the `nextDouble` method invoked on line 22. The weight output (35.4) on the last line of the figure confirms this.

```
1   import java.util.Scanner;
2   public class ScannerConsoleInput
3   {
4      public static void main(String[] args)
5    {
6      Scanner consoleIn = new Scanner(System.in);
7      String name;
8      int age;
9      double weight;
10
```

```
    System.out.print("Enter your name: ");
    System.out.println("Age: " + age + " Weight: " + weight +
                       " Name: " + name);
    System.out.print("\nEnter your age and weight on one line: ");
    age = consoleIn.nextInt();
    weight = consoleIn.nextDouble();
    System.out.print("Enter your name: ");
    consoleIn.nextLine(); // clears the enter keystroke from buffer
    name = consoleIn.nextLine();
    System.out.println("Age: " + age + " Weight: " + weight +
                       " Name: " + name);
  }
}
```

Console inputs and outputs:
Enter your name: Breanne
Enter your age: 18
Enter your weight: 125.7
Age: 18 Weight: 125.7 Name: Breanne

Enter your age and weight on one line: 5 35.4
Enter your name: Nora
Age: 5 Weight: 35.4 Name: Nora

Figure 4.15
The application **`ScannerConsoleInput`** followed by sample inputs and the corresponding outputs.

4.8 DISK INPUT AND OUTPUT: A FIRST LOOK

Unlike RAM memory, disk storage is nonvolatile, which means it retains the information stored on it when the computer system is powered down. As a result, it is used to archive data and program instructions. There are two types of disk files: text files and binary files. Information stored in binary files normally occupies less storage on the disk, and the information transfer is faster. That being said, text files are in wide use because all of the information in the file is stored as ASCII characters, which means it can be opened, read, modified, and restored using any text editor.

In this section, we will limit our discussion of disk file I/O to text files and the techniques for accessing the file's data items in the order in which they appear in the file. This type of access is called sequential access. The alternate form of access, called random *access*, allows the data items to be accessed in any order. We will extend our discussion of disk I/O in subsequent chapters.

4.8.1 Sequential Text File Input

Information stored in a text file can be sequentially read into a program using the `Scanner` class's methods presented in Table 4.5. In fact, all of the concepts discussed in Section 4.7 used to read or input data from the system console apply to sequential text file input. The one exception is the creation of the `Scanner` object.

To accept input from the system console, the object was created by passing the predefined object `System.in` to the `Scanner` class's one-parameter constructor. To accept input from a sequential text file, a `File` object is passed to the `Scanner` class's one-parameter constructor. This `File` object is created using the `File` class's one-parameter constructor that accepts a string argument containing the file's path and name. Case sensitivity in this string is ignored. The import statement `import java.io.*;` is used to access the `File` class.

The code fragment presented in Figure 4.16 reads an integer from the beginning of the file named `data.txt` resident on the root of the C drive.

```
1   File fileObject = new File("c:/data.txt");
2   Scanner fileIn = new Scanner(fileObject);
3   int score;
4
5   score = fileIn.nextInt();
```

Figure 4.16
Code fragment to read an integer from the disk file **`data.txt`** resident on the root of the C drive.

The string argument sent to the `File` class's constructor (on line 1 of Figure 4.16) contains a forward slash, which is preferred over the backslash for two reasons. First, all operating systems accept a forward slash in a path definition. Second, to use a backslash the escape sequence for a backslash (\\) would have to be used inside the string argument. Most Windows-friendly programmers often forget to code the escape sequence and code the string argument as `"c:\data.txt"`. This would result in a translation error: illegal escape character, because `\d` is not a valid escape sequence.

A more insidious error occurs when a single backslash is erroneously coded, and the character that comes after it is a valid escape character. For example, if the file name was `newData.txt`, and it was located on the root of the C drive, the following line of code would not result in a translation error on a Windows system because `\n` is a valid escape sequence.

```
File fileObject = new File("c:\newData.txt");
```

However, it would result in a runtime error indicating that the file does not exist because the `\n` would be replaced at compile time with a new line or line feed (LF) character, and the name of the file passed to the constructor would be the LF character followed by `ewData.txt`.

! TIP

Always use forward slashes (/) when specifying a file path.

Even when the forward slash is used to specify the path to the file, the file must exist or a run-time error indicating that the file does not exist will occur. If the path is not specified (i.e., just the file name and its extension is coded), the file is assumed to be inside the project folder created by the IDE or a subfolder of that folder. The exact location may be IDE-specific.

Except for lines 1 and 2 of Figure 4.16, the code used to read data from a text file is the same as the code used to read data from the system console, except that no prompts are output. We simply imagine that instead of the user typing the data into the system console in response to input prompts, the same data (character for character, line for line) was typed into a text editor and then saved to the disk file.

For example, if a person's age, weight, and name were typed into the C-drive resident text file `data.txt` whose contents are shown in Figure 4.17, then the code fragment presented in Figure 4.18 would read these values from the disk file. With the exception of lines 1 and 2, Figure 4.18 contains the same code used to read an age, weight, and name from the system console (lines 7–9 and 20–25 of Figure 4.15) with the two user prompts (lines 20 and 23) removed and the name of the Scanner object changed.

```
5      35.4
Nora
```

Figure 4.17
The data contained in the disk file **data.txt** resident on the root of the C drive.

```
1   File fileObject = new File("c:/data.txt");
2   Scanner fileIn = new Scanner(fileObject);
3   String name;
4   int age;
5   double weight;
6
7   age = fileIn.nextInt();
8   weight = fileIn.nextDouble();
9   fileIn.nextLine();
10  name = fileIn.nextLine();
```

Figure 4.18
The code fragment to read the data contained in the file shown in Figure 4.17.

To process a sequential file, Java maintains a read position pointer that is initially positioned at the first character in the file. Each time a data item is read from the file, this pointer is moved to the next item in the file. After the last item in the file has been read, the pointer is positioned at a special character called an end of file (EOF), which is automatically placed at the end of all disk files. In Chapter 5, we will discuss the importance of the addition of the EOF character to the file and how to detect when we have reached it.

There are some additional issues to consider when reading data from a text file that do not arise when performing console input. These include the need to know:

- the name and the path to the file to declare the `File` object (line 1 of Figure 4.18)
- the order of the information in the file, so the statements on lines 7, 8, 9, and 10 of Figure 4.18 are coded in the proper sequence
- the type of each piece of information in the file, so the proper `Scanner` class method can be invoked to read each piece of information

This information is described in a file specification given to the programmer by the software engineer who designed the file.

4.8.2 Determining the Existence of a File

Another issue to consider when reading data from a text file that does not arise when performing console input is how to prevent a runtime error if the data file does not exist. The `File` class contains a non-void method named `exists` that can be used to detect the existence of a file, and the `System` class contains a static method named `exit` that can be used to end a program.

The `exists` method in the `File` class returns `true` when the file exists, and the `exit` method in the `System` class has one integer parameter, which is usually passed a zero. The following code segment demonstrates the use of these two methods to bring a program to a more informative user-friendly ending when it tries to use a disk file that does not exist:

```
File fileObject = new File("c:/data.txt");


if(!fileObject.exists()) // file does not exist
{
  System.out.println("the file does not exist, the program is terminating")
  System.exit(0);
}
```

! TIP It is good programming practice to check for a disk file's existence to avoid a runtime error that is normally difficult for the user to understand.

4.8.3 Sequential Text File Output

Information can be sequentially output (written) to a text file using the `print` and `println` methods that are used to write information to the system console. In addition, the Java syntax used to format console output data, such as the spacing of the output information, moving to a new line, and specifying the precision of numeric outputs, is the same syntax used to format disk-file output. The one exception to this is the output annotation.

Output annotation is normally not included in the string sent to the methods and `print` and `println` when writing to a disk file because most disk files are read by programs, not people. When the file's data will not be processed or read by a program (perhaps the file's contents will be examined after it is printed), output annotation is included. Alternately, the reader could refer to the file's specification to identify unannotated file information.

To write to the system console, the `print` and `println` methods operate on a predefined object `System.out` attached to the system console. To write to a sequential text file, these methods operate on a programmer-defined object in the `PrintWriter` class. This object is created using two lines of code that are analogous to the two lines used to create the `Scanner` object used to perform disk input.

The `PrintWriter` object is created using the class's two-parameter constructor, which is passed to an object in the `FileWriter` class. The file's path and name is passed to the `FileWriter` object when it is created. Case sensitivity in this string is ignored. The import statement `import java.io.*;` is used to access the `PrintWriter` and the `FileWriter` classes.

The code fragment presented in Figure 4.19 creates a sequential text file named `data.txt` on the root of the C drive and then outputs the contents of the variable `score` followed by a new-line character to the beginning of the file.

```
1   FileWriter fileWriterObject = new FileWriter("c:/data.txt");
2   PrintWriter fileOut = new PrintWriter(fileWriterObject, false);
3   int score = 20;
4
5   fileOut.println(score);
```

Figure 4.19
Code fragment to write an integer to the beginning of the disk file **`data.txt`** resident on the root of the C drive.

Lines 1 and 2 of Figure 4.19 create the disk file and the object `fileOut` that is used to invoke the `println` method on line 5. A generic term used to describe the functionality of these two lines is that they create and open the file. If the file had already existed, it would have been deleted and then recreated. All the information previously written to a deleted file is lost.

Data written to a text file using the `print` and `println` methods should be thought of as being placed in the file exactly as the data would have appeared on the system console (line for line, character for character) had the methods operated on `System.out`. The only exception is that a new-line character does not appear on the system console. Rather, it causes the cursor to move to the beginning of the next line. The characters of the first data item are followed in the file by the characters of the second item, which are followed by the third, etc.

Figure 4.20 presents an application that writes a person's age, weight, and name to a sequential text file named `data.txt` and then reads the data from the file and outputs the information to the system console. The system console output produced by the program is shown at the end of the figure, and the characters written to the disk file are shown in Figure 4.21.

Lines 1 and 2 of Figure 4.20 make the `Scanner`, `File`, `FileWriter`, and `PrintWriter` classes available to the program. Their constructors are used on lines 8–12 to create objects `fileIn` and `fileOut`, which are used on lines 18–19 and lines 23–26, respectively, to write to and read from the file.

```
1    import java.util.Scanner;
2    import java.io.*;
3
4    public class DiskIO
5    {
6      public static void main(String[] args) throws IOException
7      {
8        File fileObject = new File("data.txt"); // input
9        Scanner fileIn = new Scanner(fileObject);
10       FileWriter fileWriterObject = new FileWriter("data.txt"); // output
11       PrintWriter fileOut = new PrintWriter(fileWriterObject, false);
12
13       String name = "Nora Smith";
14       int age = 5;
15       double weight = 35.4;
16
17       // write three data items to the disk file
18       fileOut.println(age + " " + weight);
19       fileOut.println(name);
20       fileOut.close();
21
22       //read the data from the disk file
23       age = fileIn.nextInt();
24       weight = fileIn.nextDouble();
25       fileIn.nextLine(); // clears New Line after a numeric from the buffer
26       name = fileIn.nextLine();
27       fileIn.close();
28
29       System.out.println("Age: " + age + " Weight: " + weight +
30                          " Name: " + name);
31     }
32   }
```

System console output

Age: 5 Weight: 35.4 Name: Nora Smith

Figure 4.20

The application **`DiskIO`** and the console output it produces.

5 35.4$\mathbf{n}^{\mathbf{l}}$Nora Smithn^{l}$\mathbf{e}^{\mathbf{of}}$
$\mathbf{n}^{\mathbf{l}}$ represents a new-line character
$\mathbf{e}^{\mathbf{of}}$ represents an end of file (EOF) character

Figure 4.21
The characters output to the file data.txt by the application **`DiskIO`**.

A `throws` clause has been added to the end of the signature of the main method (line 6). This tells the translator that the programmer is aware that some serious runtime problems (e.g., an attempt was made to read past the EOF character) could develop during the execution of the program. However, the programmer has chosen not to include code to deal with those problems. Without the throws clause, this program will not translate. We will discuss the code to deal with these problems in the next section of this chapter.

The string containing the name of the file on lines 8 and 10 does not contain a path. Therefore, the file is created inside the project folder created by the IDE. This is not always desirable but is often used in game programs because the file contains information about the game, such as the highest score achieved to date.

Line 18 writes two numbers to the file separated by a space as shown in Figure 4.21. The space is a very important part of the output. Without it, Nora's age (5) and her weight (35.4) would be adjacent to each other and would therefore be considered one number (534.4) by anyone reading the file including line 23 of the program. The result would be one of the serious runtime errors the programmer chose to ignore because a double, 534.4, cannot be parsed into the integer variable age.

Lines 20 and 27 invoke the `close` method in the `FileWriter` and `Scanner` classes. These statements release the system resources required to perform disk I/O. If they are not included in a program that performs disk input and/or output, the Java runtime environment releases the resources when the program ends. It is not only good programming practice to code them immediately after the last disk I/O statements, but in this program it is essential that line 20 be part of the program.

Here's why: During the execution of a program that writes to a disk file, the data is actually written to a RAM resident buffer. The characters stored in the buffer are written to the disk file when the buffer is full or the `FileWriter`'s `close` method is executed. This method flushes a partially full buffer to the disk file during the program's execution. Because the number of characters written by this program does not fill the buffer, eliminating line 20 from the program presents line 23 with an empty disk file from which to read. This situation causes the program to terminate in a runtime error.

When the `FileWriter` class's `close` method executes an end of file (EOF) character is added to the end of the file.

NOTE

Always invoke the `FileWriter` *class's* `close` *method after the last file output statement.*
Always invoke the `Scanner`*'s class's* `close` *method after the last file input statement.*

4.8.4 Appending Data to an Existing Text File

Data can be appended (added to the end) of a disk file by changing the second argument sent to the `PrintWriter`'s two-parameter constructor from `false` to `true`. For example, line 11 of Figure 4.20 to would be changed to:

```
PrintWriter fileOut = new PrintWriter(fileWriterObject, true);
```

When the program is run, if the file does exist it would not be deleted. (If it does not exist, it would be created.) Each execution of the program would add data to the end of the file followed by an EOF character. Figure 4.22 shows the contents of the file after three executions of the program, assuming the file did not exist before the program's first execution and the value `true` was passed to the `PrintWriter` constructor.

5 35.4**n**^{l}Nora Smith**n**^{l}5 35.4**n**^{l}Nora Smith**n**^{l}5 35.4**n**^{l}Nora Smith**n**^{l}**e**^{of}
n^{l} represents a new-line character
e^{of} represents an EOF character

Figure 4.22
The output by 3 executions of **`DiskIO`** with the file open for append.

4.8.5 Deleting, Modifying, and Adding File Data Items

Java, like most programming languages, does not contain a method to delete or modify a file data item or add a data item anywhere in the file except at its end. These operations can be accomplished by including the disk I/O methods discussed in this chapter in algorithms that perform these tasks. For example, the delete algorithm would be:

1. Read all of the file's information into RAM memory
2. Close the file
3. Delete the file
4. Recreate the file
5. Write all of the information except the item to be deleted back into to the file
6. Close the file

Because the coding of these algorithms requires knowledge of the material covered in Chapters 5 and 6, we will return to this topic in Chapter 6.

4.8.6 Redirecting the `System.in` and `System.out`

In some applications is it necessary to redirect `System.in` and `System.out` to a disk file so that output performed by the `println` or `print` methods can be written to a disk file and then input back to the program on a subsequent running of the program using the `Scanner` class's `nextLine` method.

To redirect `System.out` to a file, an instance of the class `File` is created and its constructor is passed the path to the file. Then an instance of the class `PrintStream` is created, and its constructor is passed the `File` object. Finally, the `System` class's static method `setOut` is invoked and passed the `PrintStream` object. All subsequent output to `System.out` will be written to the file. The following method redirects `System.out` to the text file data.txt located in the folder MyDocuments on the root of the C drive. The string "This is written to a disk file" is then written to the file data.txt, and the file is closed.

```
public static void redirectSystemOut() throws FileNotFoundException
{
   File file = new File("c:/MyDocuments/data.txt");
   PrintStream ps = new PrintStream(file);
   System.setOut(ps); // System.out is now the file data.txt
   System.out.println("This is written to a disk file");
   ps.close();
}
```

Similarly, to redirect `System.in` to a file, an instance of the class `File` is created and its constructor is passed the path to the file. Then an instance of the class `FileInputStream` is created, and its constructor is passed the `File` object. Finally, the `System` class's static method `setIn` is invoked and passed the `FileInputStream` object. All subsequent input performed by the methods in the `Scanner` class will be read from the file. The following method redirects `System.in` to the text file, data.txt, located on the root of the C drive. It then reads a line of text from the file, data.txt, and outputs it, and finally closes the file.

```
public static void redirectSystemIn() throws FileNotFoundException
{
   File file = new File("c:/MyDocuments/data.txt");
   FileInputStream fis = new FileInputStream(file);
   System.setIn(fis); // System.in is now the file data.txt
   Scanner consoleIn = new Scanner(System.in);
   System.out.println(consoleIn.nextLine());
   fis.close();
}
```

4.9 EXCEPTIONS: A FIRST PASS

An exception is a Java feature that a method can use to communicate to its invoker that an unexpected event has occurred during the method's execution when the method does not contain code to deal with it. When the event is one that Java deems serious, a `throws` clause must be added to the signature of the method in which the invocation is coded, or instructions to deal with the event must be added to the code block that contains the invocation statement.

The former approach was taken in the program that appears in Figure 4.20 on line 6. The `Scanner` and `FileWriter` class constructors invoked on lines 9 and 10 are methods that can encounter serious unexpected events during their execution. Therefore, a `throws` clause was added to the signature of the main method (line 14) because the main method contains these two invocations. In this section, we will cover a brief introduction to the alternative to the `throws` clause: adding instructions to deal with the event in the code block that contains the invocation statements. Chapter 10, "Exceptions, A Second Pass" contains a more in-depth discussion of exceptions.

As the word "throws" implies, a baseball analogy was used in the selection of the Java keywords associated with exceptions, and the analogy is helpful in gaining an understanding of exceptions. Imagine that when the serious event occurs during a method's execution the method says, "I take *exception* to that event, and I am not going to continue executing. My last action will be to let my invoker know of this problem by *throwing* an exception object back to it."

If the invoker wants to deal with the problem, its code block *catches* the exception object and deals with the problem. Otherwise it *throws* the exception object on to the Java runtime environment. The term `throws` is a Java keyword we have already used (line 6 of Figure 4.20) when we did not want to deal with an unexpected problem. Two other keywords, **`try`** and **`catch`**, are used when we want to deal with the problem.

Each of these keywords begins a code block, and the `try` code block is always coded immediately before the `catch` code block. The following code fragment is a template for a `try` statement that will catch a thrown `IOException` object. As shown in the template, the type of exception object caught is coded in a parameter list after the keyword `catch`.

```
try
{
  // the code containing the method invocations and other statements
}
catch(IOException e)
{
  // the statements to deal with the unexpected events
}
```

The statements that invoked the methods that could throw the exception object must be coded inside the `try` block. Other statements can be included in the `try` block. Effectively, you are trying these invocation statements to see if the methods they invoke throw an exception object.

When an exception object is thrown by a method invoked inside the `try` block, the remainder of the statements in the `try` block do not execute, and execution passes to the first statement in the `catch` block. If an exception is not thrown, the statements in the `try` block complete their execution, and the `catch` block statements are not executed. In either case, the statements following the `catch` code block executes after the `try` block or the `catch` block completes execution.

The use of the template is illustrated in the following code fragment. It attempts to read the value stored in a disk file into the variable `score` and catches the `IOException` object thrown by the `Scanner` class's constructor when this constructor encounters a problem.

```
int score;
try
{
  File fileObject = new File("data.txt"); // input
  Scanner fileIn = new Scanner(fileObject);

  score = fileIn.nextInt();
  fileIn.close();
}
catch(IOException e)
{
  System.out.print ("The score could not be read from the disk file,");
  System.out.println(" but the game will continue.");
}
//rest of the game's statements
```

If the reading of the score was essential to the continuation of the program, the second statement in the `catch` block would be replaced with the following two statements to terminate the program's execution:

```
System.out.println(" the program is terminating.");
System.exit(0);
```

The program in Figure 4.23 illustrates the use of disk input and output in a game program and the use of exceptions to deal with unexpected disk I/O problems. It is the same program presented in Figure 4.13, modified to keep track of the highest game score ever achieved. When the game is over, this score is read from a disk file. If a new high score was not achieved, the game player is informed and encouraged to keep practicing. Otherwise, the new high score is written to the disk file and the game player is congratulated.

```
import edu.sjcny.gpv1.*;
import java.awt.*;
import java.util.Scanner;
import java.io.*;
//illstrates basic exceptions

public class ExceptionBasics extends DrawableAdapter
{
  static ExceptionBasics ge = new ExceptionBasics ();
  static GameBoard gb = new GameBoard(ge, "Exception Basics");
  static BoxedSnowman s1 = new BoxedSnowman(300, 200, Color.GREEN);
  static BoxedSnowman s2 = new BoxedSnowman(30, 100, Color.BLACK);
  static int score = 0;
  static int count = 10;

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }
```

```java

  public void draw(Graphics g) // a call back method
  {
    int w = 40;
    int h = 77;
    int s1X, s1Y, s2X, s2Y, temp;

    s1X = s1.getX(); s1Y = s1.getY();
    s2X = s2.getX(); s2Y = s2.getY();
    g.setColor(Color.BLACK);
    g.setFont(new Font("Arial", Font.BOLD, 18));
    g.drawString("Time remaining: " + count, 260, 50);

    if(count == 0) // the game is over
    {
      g.setColor(Color.BLACK);
      g.drawString("Game Over", 205, 70);
      g.drawString("Have a Good Day", 175, 90);

      try
      {
        int highScore;
        File fileObj = new File("HiScore.txt");
        Scanner fileIn = new Scanner(fileObj);
        highScore = fileIn.nextInt();
        fileIn.close();

        if(score >= highScore) // a new high score
        {
          g.drawString("Great, Your Score is the Highest Ever.," +
                       "It Will Be Saved", 10, 110);
          FileWriter fileWriterObj = new FileWriter("HiScore.txt");
          PrintWriter fileOut = new PrintWriter(fileWriterObj, false);

          fileOut.println(score);
          fileOut.close();
        }
        else // not a new high score
        {
          g.drawString("Best Score is: " + highScore +
                       ", Keep Practicing", 110, 110);
        }
      }
      catch(IOException e)
      {
        g.drawString("Problems With High Score File", 120, 110);
      }
    }
```

```
      else if( !(s2X > s1X + w || s2X + w < s1X || s2Y > s1Y + h ||
                 s2Y + h < s1Y) && s1.getVisible() == true)
      {
        score = score + 1;
        s1.setVisible(false);
      }
      else if( s2X > s1X + w || s2X + w < s1X || s2Y > s1Y + h ||
                s2Y + h < s1Y) // no collision
      {
        if(s1.getVisible() == false) // not visible
        {  temp = s1.getX();
           s1.setX(s1.getY());
           s1.setY(temp);
           s1.setVisible(true);
        }
      }

      s2.show(g);
      if(s1.getVisible() == true)
      {
        s1.show(g);
      }
        g.setColor(Color.BLACK);
        g.drawString("Score: " + score, 150, 50);
    }

    public void keyStruck(char key) // a call back method
    {
      int newX, newY;

      switch (key)
      {
        case 'L':
        {
          newX = s2.getX() - 2;
          s2.setX(newX);
          break;
        }
        case 'R':
        {
          newX = s2.getX() + 2;
          s2.setX(newX);
          break;
        }
        case 'U':
        {
          newY = s2.getY() - 2;
          s2.setY(newY);
          break;
```

```
117       }
118       case 'D':
119       {
120         newY = s2.getY() + 2;
121         s2.setY(newY);
122       }
123     }
124   }
125   public void timer1() // a call back method
126   {
127     count = count - 1;
128     if(count == 0)
129     {
130       gb.stopTimer(1);
131     }
132   }
133 }
```

Figure 4.23
The **`ExceptionBasics`** Application: A decision and exceptions case study.

When the application is launched, two snowmen, one wearing a black hat and the other wearing a green hat, appear on the game board below the game's score and remaining time (Figure 4.24a). The game begins when the player clicks the Start button on the game board. The objective of the game is to make the two snowmen collide as many times as possible before time runs out using the keyboard cursor-control keys to move the black-hat snowman. Each time they collide, a point is awarded, and the green-hat snowman disappears. It reappears at a new location after the black-hat snowman has been moved to a location such that the two snowmen are no longer in a collision state.

The changes to the program are the additions of the lines 3–4 that make the `Scanner`, `File`, `FileWriter`, and `PrintWriter` classes needed to perform disk I/O available to the program, the elimination of the `throws` clause in the `main` method's signature and the addition of lines 40–67 that perform the disk I/O. Figure 4.24 presents several outputs produced by the program under various game conditions.

The signature of the main method (line 16) does not contain a `throws` clause because the disk I/O is performed inside the code block of a `try` statement (line 39). Line 44 reads the highest score ever achieved from the disk file `HiScore.txt` using the `Scanner` object `inFile` created by lines 42–43. Then, the file is closed (line 45). Normally, the programmer would use a text editor to create the file and store a score of zero in it as part of the program's development. Because the file's path is not specified on line 42, the file must be stored inside the project folder created by the IDE.

When a new high score is achieved, as determined by line 47, line 49–50 informs the game player of this achievement (Figure 4.24b). The new high score is written to the disk file (line 54) using the `PrintWriter` object `fileOut` created on lines 51–52, and the file is closed (line 55). Because the second argument sent to the `PrintWriter` constructor on line 52 is `false`, the file containing the old high score is deleted and recreated before the new high score is written to it. (The new high score is not appended to the file.)

When a new high score is not achieved, lines 59–60 of the `if-else` statement that begins in line 47 produces the output shown in Figure 4.24c. If a problem occurs during the disk I/O performed inside of the `try` block, execution of the `try` block is terminated, and the `catch` block (lines 63–66) executes producing the error message at the end of the text output shown in Figure 4.24d.

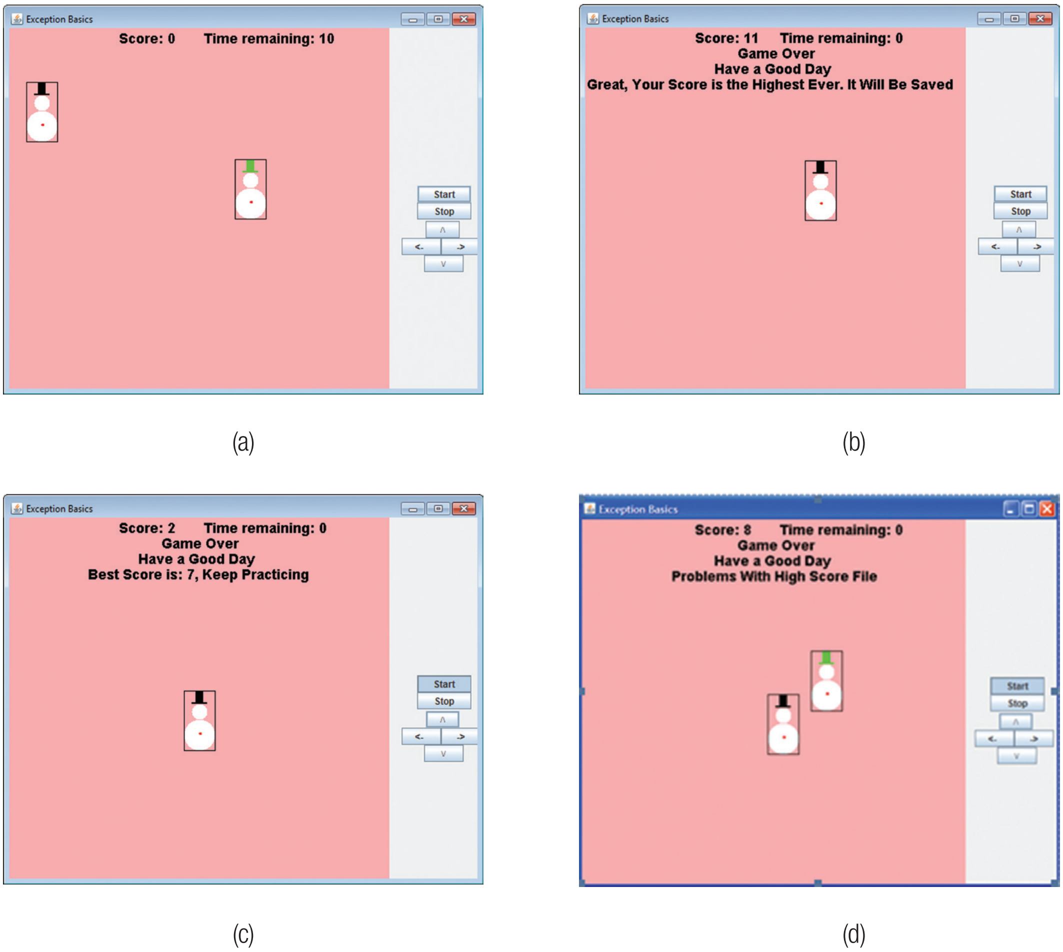


Figure 4.24
Outputs produced by the application **`ExceptionBasics`**.

4.10 CHAPTER SUMMARY

Ordinarily, a Java program executes its statements sequentially. The `if`, `if-else`, and `switch` statements are used to alter this sequential path of execution by selecting which statement, or group of statements, to execute next. When the resulting decisions of an `if` or `if-else construct` effect more than one statement, these statements must be coded inside a code block.

Both the `if` and `if-else` statements evaluate Boolean expressions to determine whether to execute or skip the statements included in their code blocks. Boolean expressions use the logical and relational operators (all of which have a precedence order associated with them) and evaluate to `true` or `false`. These statements may be nested, which allows them to test for several conditions, or several conditions can be tested by one statement using compound Boolean conditions. The `if` statement is used to decide to skip or execute one group of statements, and the `if-else` statement is used to decide which of two mutually exclusive groups of statements to execute.

Although all uses of the `switch` statement can be coded using an equivalent set of nested `if-else` statements, the use of the `switch` statement makes our programs more readable when the decision to be made is based on one or more discrete values. The values can be references to a strings or primitive values whose type is `byte`, `short`, `char` or `int`, or the values of an enumerated type.

Most often, a `break` statement is used to prevent the cases coded below the one that is equivalent to the current value of the switch variable from executing. A `default` clause can be included at the end of the statement that will execute when the value of the switch variable does not match any of the statement's cases. Decision statements have many applications to game programming, such as controlling the value of a timer, increasing a score when an event occurs, testing to see when there is a collision between objects, and determining which keystroke has been entered and responding to it.

The `String` class provides methods for comparing `String` objects because they cannot be compared using the relational operators. The equality operator compares the contents of variables, but `String` reference variables contain the address of the strings they refer to, not the strings themselves. Therefore, to compare strings, the `String` class provides methods such as `equals` and `compareTo`, which make case sensitive comparisons, and `equalsIgnoreCase` and `compareToIgnoreCase`, which ignore case sensitivity.

Disk I/O is useful for storing game data, such as the highest score achieved, and any other type of data that must be retained after a program ends. When data is stored in RAM buffers, it is volatile and is not preserved from one game to the next. In contrast, when data is stored in a text file, it can be read and compared each time the game is played. The constructors in the `File` and `Scanner` classes, and the constructors in the `FileWriter` and `PrintWriter` classes, can be used to "attach" `Scanner` and `PrintWriter` I/O objects to a file. Text output can then be sent to the file using the methods in the `PrintWriter` class, and information can be read from the file using the `Scanner` class's methods. Methods in the scanner class can also be used to accept input from the system console. The `File` class method, `exists`, can determine if a file exists before attempting to use it, and it is always good programming style to close a file after using it.

Disk I/O often causes errors such as when code attempts to access a file that does not exist or whose pathname is incorrect. This causes an exception error to be generated, which disrupts the normal flow of a program. Java provides two exception-related constructs, called `try` and `catch` blocks. When an exception occurs within the code of a `try` block, the program's execution path is transferred into the code of the `catch` block, which is designed to process (handle) and recover from runtime exceptions. A `System` class method, `exit`, can be used inside a `catch` block to terminate a program gracefully.

Knowledge Exercises

1. The variables `i`, `j`, `k` and `m` have been declared as: **int** `i` = 10; **int** `j` = 20; **int** `k` = 30; **int** `m` = 40. Evaluate the following as true or false:
   a) `i <= j`
   b) `k == 30`
   c) `i != j && j >= k`
   d) `i != j || j >= k`
   e) `(i <= k && k >= m) || (j * 2 == m && k > j)`
   f) `m <= k || i + j + k >= m`
2. What is the normal default execution sequence (path) of all Java programs?
3. What are the two types of statements available in Java to alter the default execution path?
4. Write the Java code to output the contents of the variable `myBalance` when it stores the value 10.0 to the system console.
5. Modify the Java code in Question 4 by adding a statement to output *my balance is not 10.0* when the memory cell `myBalance` does not store `10.0`.
6. Write an `if` or `if-else` statement to perform each of these tasks:
   a) Add 5 to a grade if the grade is greater than 75
   b) Produce the output *buy tickets* if the cost is less than $150.00, otherwise output *too expensive*. Then use a conditional expression to produce the same output.
   c) Output the value stored in the variable `GPA` to the system console if the `String` object `name` contains the string `Anna`
   d) For the strings referenced by `s1` and `s2`, if `s1` comes before `s2` in alphabetical order, output *in alphabetical order* otherwise output the message *not in order*.
7. True or false:
   a) An `if` statement must contain one Boolean expression.
   b) An `if-else` statement must contain two Boolean expressions.
   c) An `if` statement must contain a statement block.
   d) The code block of an `if` statement executes when its Boolean condition is `true`.
   e) The code block of an `if` statement can contain another `if` statement.
   f) The code block of an `else` clause cannot contain another `if` statement.
   g) An `if` or `if-else` statement may be nested within another `if` statement's code block.
   h) Java contains an `if-else-if` statement.
8. The string `s1` just received input from an input dialog box. Give the statement to output *OK* to the system console when the user enters *Stop Sign* (case sensitive).
9. True or false:
   a) A `switch` statement must contain a default clause.
   b) A `switch` statement can have multiple cases.

c) The choice values of a `switch` statement can be strings.

d) A `switch` statement must contain at least one `break` statement.

e) A `switch` statement is normally used to determine a choice between several alternatives.

f) Every `switch` statement can be written as equivalent `if-else` statements.

g) A sequence of `if-else` statements can always be written as an equivalent `switch` statement.

10. Write the `switch` statement to output the menu selection stored in the string variable `item`, assuming the choices are *`Hamburger`*, *`Taco`*, or *`BLT`* (use system-console output).

11. Write the equivalent `if-else` statements to output the menu selections given in Question 10.

12. What API class must be imported into your program to accept input from the system console?

13. Give all of the statements, excluding import statements, to:

a) accept the year of a person's birth input from the system console (include a prompt)

b) accept a person's name input from the system console (include a prompt)

14. What is the advantage of saving information in disk files versus saving the information in main memory?

15. True or false:

a) Text files can be viewed using the program Notepad.

b) Text files cannot be printed on a printer.

c) By convention, text files end with the extension .txt

d) It is best to use two forward slashes to specify the path name where the file is located.

16. Write all of the import statement(s) necessary to perform disk I/O.

17. Give all of the statements, excluding import statements, to:

a) read the year of a person's birth from the disk file `Dates.txt` stored on the root of the E drive

b) read a person's age and name from the disk file `Names.txt` stored on the root of the E drive

c) read the year of a person's birth from the disk file `bDays.txt` stored on the root of the E drive

18. Give all the statements necessary to append the contents of the variables `myBalance` and `yourBalance` to the disk file `Balances.txt` stored on the root of the C drive.

19. Give all the statements necessary to output the contents of the variables `myBalance` and `yourBalance` to the disk file `Balances.txt` stored on the root of the C drive. If the file already exists, delete it before performing the output.

20. Give all the statements necessary to determine if the file `Data.txt` exists on the root of the C drive, and output *The File Exists* to the system console if it does.

21. Write the statement needed to close the file attached to the scanner object `inputFile`.

22. Briefly discuss how the `try` and `catch` block can be used to handle exceptions detected by methods invoked within a program.

Programming Exercises

1. Write a program to ask a user to input two strings from the system console. If the strings are identical output *Stings Identical* to the system console. Otherwise output them in alphabetical order.

2. Write a program for a travel agency, which presents the user with the following menu as a console input prompt:

   Where do you want to vacation?

   Enter: 1 for Disney World, 2 for Las Vegas, 3 for Paris or 4 for Alaska

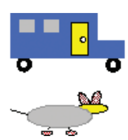
   After accepting the customer's numeric response from the system console, use either an `if-else` or a `switch` statement to output two destination-appropriate messages to the text file `vacation.txt` (for example, if the user chose 4 for Alaska, you might want to output the messages *Bring a warm jacket and enjoy Alaska* and *Say "Hello" to Frosty for me*). Feel free to add bells and whistles such as adding a welcome message or adding additional destinations.

3. Write a program to ask a user to enter a student name, major, and GPA from the system console. If the GPA is greater than 3.5, set a Boolean variable, `honors`, to `true`, otherwise set it to `false`. Create a text file called `StudentInfo.txt` and output the name, major, GPA, and the student's honors status to four separate lines of the file.

4. Extend the program described in Programming Exercise 4 to ask the user the name of the file in which to store the data. After writing the data to a text file with that name, add the phrase *End of Student Record* as the last line of the file. Then, read five lines store in the file and output them on five separate lines to a message box with the appropriate annotation. Before reading the data, ask the user the name of the file from which to read. Use a `try` and a `catch` block to output the message *problems opening or reading the file* when an `IOException` is thrown.

5. Write a graphical game application that contains a class named `RV` whose objects are the recreational vehicle designed and digitized as described in Knowledge Exercises 20 and 21 of Chapter 3. When the application is launched, the RVs appear on the screen. The game player is given 10 seconds to move any part of the RV beyond the top, bottom, right side, and left side of the game board using the keyboard cursor control keys. The game begins when the game player clicks the Start button and ends when the time expires or some part of the RV has moved beyond all four boundaries of the game board. During the game, a countdown of the time remaining should be displayed at the top of the game board, and the countdown should stop at the end of the game.

6. Write a graphical game application that contains a class named `RV` whose objects are the recreational vehicles designed and digitized as described in Knowledge Exercises 20 and 21 of Chapter 3. The application should also contain a class named `Mouse` whose objects are designed and digitized in a similar manner. When the application is launched, one RV



and one mouse appear on the screen at different random locations. The user is given 10 seconds to move the mouse to the RV using the keyboard's cursor control keys. The game begins when the game player clicks the Start button and ends when the time expires or the mouse has collided with the RV. During the game, a countdown of the time remaining should be displayed at the top of the game board, and the countdown should stop at the end of the game. The game begins when the game player clicks the Start button and ends when the time expires or some part of the mouse has collided with the RV.

7. Write the game application described in Programming Exercise 6 modified to include three RVs at different locations. In this version of the game, the game player has to make the mouse collide into all three RVs and the RVs disappear when the mouse collides with them. The player's score will be the time remaining after all the RVs have disappeared. A record of the lowest score ever achieved will be kept in the disk file `LowScore.txt`.

8. Using the skills developed in this chapter, continue the implementation of the parts of your game (specified in Preprogramming Exercise 1 of Chapter 1) that require cursor-key motion control, disk I/O, collision detection, and stopping a time countdown. To test the collision detection, you will have to add a class to your application that implements your second type of game piece.

CHAPTER 5

REPEATING STATEMENTS: LOOPS

In this chapter

In this chapter, we will learn the techniques used to repeat the execution of a designated group of statements an unlimited number of times, which gives us the ability to perform a significant amount of processing with just a few repeated statements. Not only does this reduce the time and effort required to produce a program, but it also allows us to utilize algorithms whose implementation require the use of these repetition, or loop, statements. We will discuss the syntax and execution path of Java's three how to nest these statements. Our knowledge of these statements will be expanded in Chapter 6, which covers the concept of arrays, because loops are used to unlock the power of arrays.

We will learn why repetition statements are an integral part repetition statements, consider which one is best suited for particular applications, and learn of two fundamental algorithms, summing and averaging, and how to generate a repeatable sequence of pseudorandom numbers using loops and the methods in the class `Random`. In addition, we will extend our knowledge of numeric formatting introduced in Chapter 2 and learn to produce output consistent with any of the world's currency systems.

After successfully completing this chapter you should:

- Understand the syntax and execution path of Java's `for`, `while`, and `do-while` repetition statements
- Know which statement to use for a particular application
- Understand why a `for` loop is an automatic counting loop
- Understand the role of sentinels in repetition statements and their use in `while` and `do-while` loops

- Be able to explain the totaling and counting algorithms and the role loops play in their implementation
- Know how to generate a set of random numbers using the methods in the class `Random` and loop statements
- Be able to use the `NumberFormat` class's methods to format currency output in a local specific format
- Be able to use the `DecimalFormat` class's methods to format numeric output with leading/trailing zeros and comma separators and display a numeric value as a percentage or using scientific notation

5.1 A SECOND ALTERNATIVE TO SEQUENTIAL EXECUTION

Often, the proper execution path of a program's statements requires that a sequence of instructions be executed several times. For example, a program accepts three input deposits and adds them to a bank balance after each input. In this case, the input statement and the arithmetic statement to add the input deposit to the bank balance would be repeated three times.

One alternative would be to code one input and one arithmetic statement, copy and paste them into the program two more times, and then execute the three groups of statements sequentially. Another alternative would be to enclose one input and one arithmetic statement in a repetition statement's code block, which is repeated three times. Although both approaches would produce the same result, the second alternative is most often preferred, especially when the statements are to be repeated a large number of times. Not only does this approach save coding time, but it also improves the readability of our programs by significantly reducing the length of the program, and, more importantly, making it obvious to the reader that the statements are being repeated.

A repetition statement is most often referred to as a *loop* statement. The term loop comes from an aircraft "loop" maneuver often performed at air shows during which the aircraft repeatedly travels in a vertical circle. Figure 5.1 illustrates the maneuver and programming analogy.

Like many programming languages, Java provides three repetition or loop statements: the `for`, the `while`, and the `do-while` statements. While there is the possibility for significant overlap in

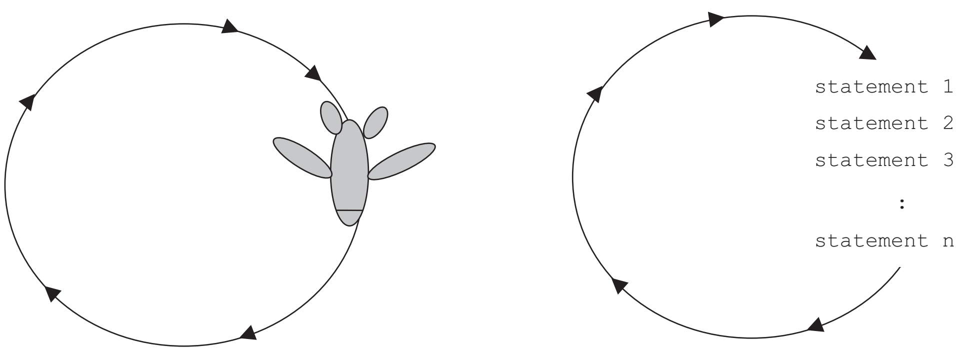


Figure 5.1
Airplane and programming loops.

the use of these three statements in our programs, good coding practice and ease of use greatly narrow the choice of which statement to use in a particular context. One of the objectives of this chapter is to specify clear criteria for when each of these three statements is best used in our programs. We will begin our study of repetition statements with the `for` statement.

5.2 THE `FOR` STATEMENT

The `for` statement is often called an automatic counting loop. It is most often used when we know how many times to repeat the loop's statements. In some cases, this is known at the time the program is written; for example, a program that always processes three deposits. In other cases, the number of times the loop is to execute is specified, or determined, during the program's execution. For example, before entering deposits the program users are asked to enter the number of deposits they will be making into their bank account during this execution of the program. The criterion common to both of these alternatives is that *before* the loop executes the number of repetitions is known. When this is the case, the `for` loop is the best repetition statement to use.

5.2.1 Syntax of the `for` Statement

The left side of Figure 5.2 shows an example of a `for` statement containing a group of statements that will executes its statement block three times. The meaning of the statement and its execution path are illustrated on the right side of Figure 5.2. The integer variable `i` is called the loop or

```
int  i;
for (i=1; i<=3; i=i+1)
{
   //statement(s) to be repeated
}
```

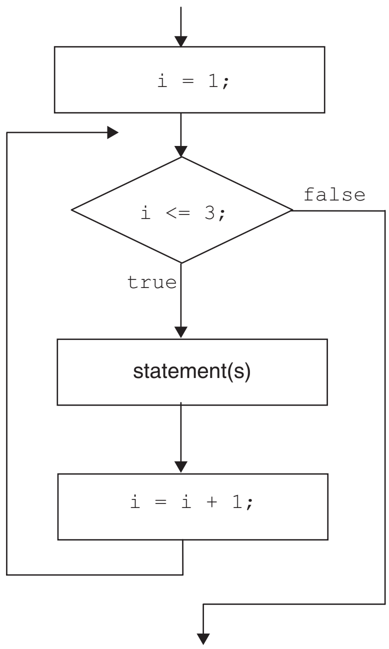


Figure 5.2
A **`for`** loop that executes three times and its execution path.

loop control variable. The statement(s) enclosed inside the braces are called the loop's code block, or the *body of the loop*. They are said to be inside the loop. These are the statements to be repeated.

When a loop is correctly written, the loop variable is initialized, tested, and changed. In a `for` loop, all three of these actions are coded within the `for` statement's parentheses. Referring to the left side of Figure 5.2, the statement `i=1;` sets the initial value, `i<=3;` tests to see if `i` has reached its terminating value or if the loop should continue, and `i=i+1;` changes or increments the value of the loop's control variable.

Referring to the items enclosed inside the parentheses after the keyword `for`, the code:

- `i=1`; is called the initialization expression
- `i<=3;` is called the condition to continue expression or continuation condition
- `i=i+1;` is called the increment

The initialization expression is an assignment statement. As shown on the top-right side of Figure 5.2, this assignment statement always executes once to initialize the loop variable just before the loop begins. The condition to continue expression is a Boolean expression involving the loop variable, which executes at least once. The loop body is repeatedly executed while this Boolean expression is `true`. If the Boolean condition is `false` when the loop begins, the statements in the loop body are not executed. The increment is an assignment statement. The statement is used to change the loop variable after the statements in the loop body are executed. When the increment is one, the equivalent coding `i++`; is commonly used.

The loop shown in Figure 5.2 executes its statement body three times. When the `for` statement begins, the loop variable `i` is initialized to 1. The condition to continue (`i <= 3`) evaluates to `true` (`1 <= 3`), and the statements in the loop body execute for the first time. The loop variable is then incremented to 2, the condition is tested and is still `true` (`2 <= 3`), and the statements execute a second time. The loop variable is then incremented to 3, the condition is still `true` (`3 <= 3`), and the statements execute a third time. Finally, the loop variable becomes 4, the condition (`4 <= 3`) is `false`, and the loop ends. After the loop ends, the statement immediately following its close brace executes.

The following code fragment contains a `for` statement that executes its loop body 500 times:

```
int i;
for(i=1; i<=500; i=i+1)
     {
       //statement(s) to be repeated
     }
```

The most common errors made when coding the `for` statement are:

- placing a semicolon after the close parenthesis
- neglecting to code the semicolon after the initial condition or after the condition to continue, both of which result in a translation error
- neglecting to code the open and close braces around the statements when more than one statement is to be repeated

- modifying the loop variable within the body of the loop which alters the automatic counting

When a semicolon is coded after the close parenthesis, the statement is syntactically correct, however, none of the statements that would have normally formed the loop body are considered to be part of (inside) the loop. They default to sequential execution and each statement executes once. When braces are not coded, the statement is also syntactically correct, however, the first statement after the `for` statement is the only statement considered to be part of the loop. Regardless of the indentation used, it is the only statement repeated.

The generalized syntax of the `for` statement and its execution path are shown at the top and bottom of Figure 5.3, respectively.

```
for(initialization expression; condition to continue; increment)
{
   //statement(s) to be repeated
}
```

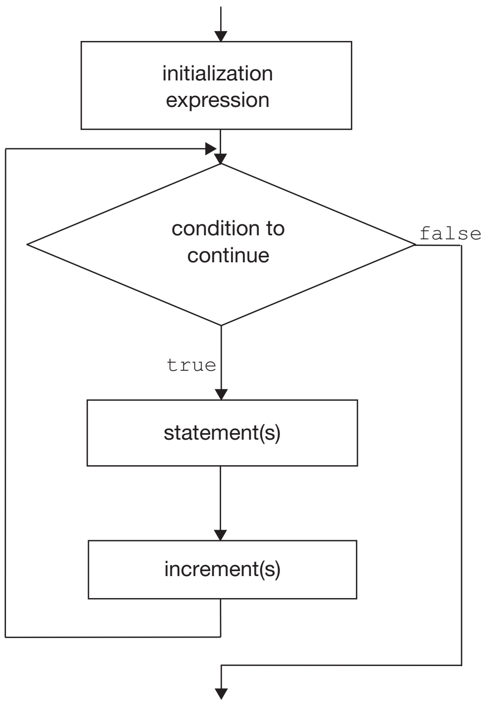


Figure 5.3
The generalized syntax of the **`for`** statement and its execution path.

In addition to the loop variable, the initialization expression, test to continue, and the increment can all contain other variables that can be used to adjust the flow of the statement at runtime. For example, the following code fragment outputs the values in the five times table from 10 to 50 on one line:

```
int i;
int beginValue = 10;
```

```
int endValue = 50;
int tableValue = 5;

for(i=beginValue; i<=endValue, i=i+tableValue)
{
System.out.print(i + " ");
}
```

Figure 5.4 presents a console application named `ForLoopCounting` that utilizes this feature of the statement to count from a user input starting value to a specified ending value, by a specified increment. The bottom part of the figure gives the user prompts and inputs and the corresponding outputs generated by the program.

The input starting and ending values, and the increment to count by, are parsed into the variables `start`, `end`, and `increment` on lines 10, 12, and 14 of Figure 5.4. These variables are used in the initialization expression, condition to continue, and increment of the `for` statement that begins on line 18. As indicated by the input and output at the bottom of Figure 5.5, the program user inputs 3 as a starting value, 27 as an ending value, and an increment of 5. After the value 23 is output, the loop variable `i` becomes 28 (= 23 + 5). Because 28 is not less than or equal to the ending value 27, the loop ends and 28 is not output.

```
1    import javax.swing.*;
2
3    public class ForLoopCounting
4    {
5      public static void main(String[] args)
6      { int start, end, increment;
7        String input;
8
9        input = JOptionPane.showInputDialog("Enter the starting value:");
10       start = Integer.parseInt(input);
11       input = JOptionPane.showInputDialog("Enter the ending value: ");
12       end = Integer.parseInt(input);
13       input = JOptionPane.showInputDialog("Count by?: ");
14       increment = Integer.parseInt(input);
15
16       System.out.println("Counting from " + start + " to " + end +
17                          " by " + increment + "s:");
18       for(int i=start; i<=end; i=i+increment)
19       {
20         System.out.println(i);
21       }
22     }
23   }
```

Input prompts and user inputs:

Enter the starting value: 3

Enter the ending value: 27

Count by?: 5

```
Outputs:
Counting from 3 to 23 by 5s:
3
8
13
18
23
```

Figure 5.4
The application **`ForLoopCounting`** and typical inputs and outputs.

Line 18 presents a feature of the `for` statement we have not previously discussed. It declares the loop variable `i` as part of the initialization expression by proceeding its assignment statement with the keyword `int`. When this is done, the scope of the loop variable is limited to the `for` statement and its statement body. The loop variable cannot be used by statements that follow the loop or by statements that precede the loop. After the loop ends, the Java memory manager reclaims the storage assigned to the loop variable, and the variable's lifetime is said to be over. If another variable named `i` had been declared in the program before or after the loop statement, all references to the variable `i` inside the `for` statement (lines 18-21) would refer to the loop variable, not the variable declared outside the loop. This feature ensures that the loop will count correctly.

A `for` loop can also be used to count down to an ending value. In this case, the loop variable is initialized to the starting countdown value, and it is decremented each pass through the loop. The statement's Boolean condition checks to see if the ending value is reached. The following code fragment counts down from ten to zero:

```
for(int i= 10; i>= 0; i--)
{
  System.out.println(i);
}
```

In general, the initialization expression and the increment can contain more than one assignment statement. When this is the case, they are separated with commas as shown in code fragment below. The loop's execution path is shown in Figure 5.5.

```
int i, j, k;
for(i=1, j=10, k=4; i<= 3; i=i+1, j=j+3, k=k-1)
{
  //statement(s) to be repeated
}
```

5.2.2 A `for` Loop Application

Figure 5.6 shows a graphical application that draws the first row of a checkerboard on a light-gray-colored game board as shown in Figure 5.7. The program uses a `for` loop to draw the eight checkerboard squares and then uses another `for` loop to draw a red checker on the row's black squares.

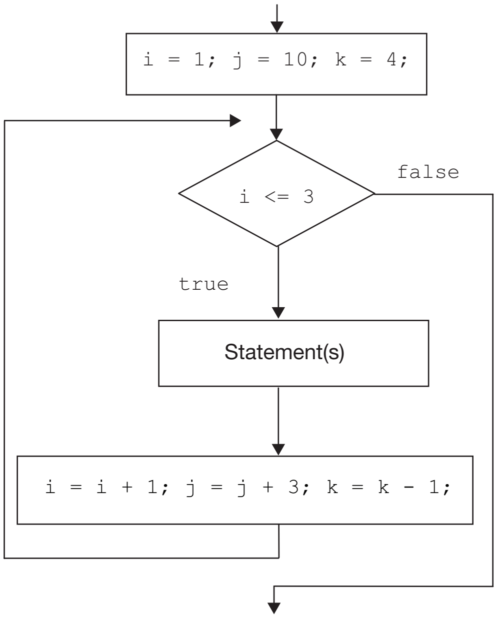


Figure 5.5
Execution path of a **for** loop containing multiple initialization-expression and increment-assignment statements.

Inside the `draw` method, line 26 of Figure 5.6 uses the game environment's `setBackground` method to change the game board's background color to light gray. Then line 29 begins a `for` loop that executes eight times. During each iteration of the loop, a black or red checkerboard box is drawn (line 36) using the current drawing color. The first statement in the loop body (line 31) sets the current color to `firstColor` (black), then line 32 uses the modulus operator to determine if the loop variable, `col`, is even (`col % 2 == 0`). If it is, the current color is changed to `secondColor` (red), which causes the even checkerboard boxes (2, 4, 6, and 8) to be drawn in red. The counting algorithm, whose increment is the width of the checkerboard boxes, is used on line 37 to calculate the x location of the next checkerboard box to be drawn.

Line 42 begins a second `for` loop that draws a red checker (line 45) on the black checkerboard boxes. Before the loop begins, the current drawing color is set to red (line 41). Because the `for` statement's increment adds 2 to the loop variable `col`, this variable stores the column numbers 1 (`firstCheckerCol`), then 3, 5, and 7. These are the column numbers of the black boxes, which are used on line 44 to calculate the x location of each column's checker. In this calculation, one is subtracted from the column number `col` before it is multiplied by the box width because column 1's checker should be drawn at an x value of 20.

```
1   import java.awt.*;
2   import edu.sjcny.gpv1.*;
3
4   public class CheckerBoardRow extends DrawableAdapter
5   {
6      static CheckerBoardRow ge = new CheckerBoardRow();
7      static GameBoard gb = new GameBoard(ge, "Checker Board Row");
8
9      public static void main(String[] args)
10     {
11       showGameBoard(gb);
12     }
13
14     public void draw(Graphics g)
15     {
16       int boxX = 12;
17       int boxY = 50;
18       int boxWidth = 60;
19       int boxHeight = 53;
20       int checkerX = 20;
```

```
        int checkerY = 55;
        int firstCheckerCol = 1;
        Color firstColor = Color.BLACK;
        Color secondColor = Color.RED;

        gb.setBackground(Color.LIGHT_GRAY);

        //Draw the Checker board boxes
        for(int col = 1; col <= 8; col++)
        {
          g. setColor(firstColor); //black
          if(col % 2 == 0)
          {
            g. setColor(secondColor); //red
          } //end if
          g.fillRect(boxX, boxY, boxWidth, boxHeight);
          boxX = boxX + boxWidth;
        } //end for loop

        //Draw the Red checkers
        g.setColor(Color.RED);
        for(int col = firstCheckerCol; col <=8; col= col + 2)
        {
          checkerX = 20 + (col - 1) * boxWidth;
          g.fillOval(checkerX, checkerY, 40, 40);
        }
      }
    }
```

Figure 5.6
The application **CheckerBoardRow**.

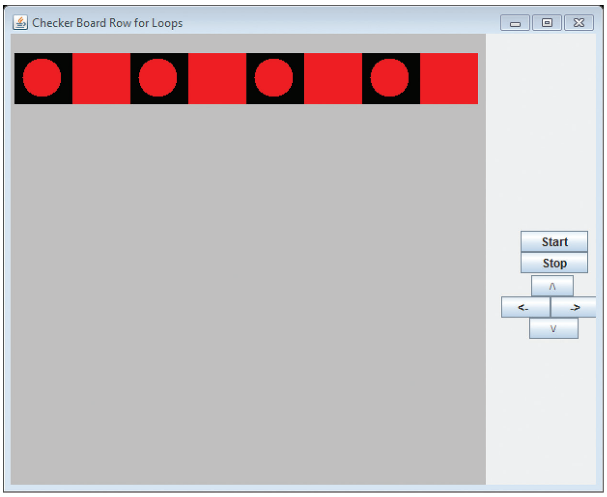


Figure 5.7
The output from the application **CheckerBoardRow**.

5.2.3 The Totaling and Averaging Algorithms

The totaling or summation algorithm, like the counting algorithm, is a fundamental algorithm of computer science. Both of these algorithms are used in most programs. As the totaling algorithm's name implies, it is used to calculate a total, or sum, of a group of values. For example, it could be used to calculate the total of a group of deposits input to a program, and once the total is known, the average deposit can be easily determined by dividing the total by the number of deposits.

The totaling algorithm is very similar to the counting algorithm, except that the counting algorithm adds (or subtracts) a *constant* increment to the current value of the counter every time it is executed, and the totaling algorithm adds the *next item* to be totaled to the current value of the total every time it is executed:

counting algorithm: `count = count ± constantIncrement;`
totaling algorithm: `total = total + newItem`

If we were adding a group of deposits, the variable `newItem` would contain the next deposit. As was the case with the counting algorithm, the name of the variable on the left side of the assignment operator can be any valid variable name. Similarly, the same variable name must be used on the right side of the assignment operator, and the name should be representative of what it stores. For example, when calculating a new bank balance:

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

The chosen variable is generically referred to as the totaling or summation variable. Before the algorithm is used, the variable is set to an initial value, which is the current (beginning) value of the sum. For example, it could be the bank balance before the new deposits are made. Often, this value is zero. As is the case with the counting algorithm, once the initial value is set, the summation algorithm is repeatedly executed.

The following code fragments add up the integers from one to four:

```
int sum = 0;                              int sum = 0;
sum = sum + 1;                            for(int i = 1; i <= 4; i = i + 1)
sum = sum + 2;          OR                {
sum = sum + 3;                               sum = sum + i;
sum = sum + 4;                            }
```

The contents of the memory cell `sum` would progress from 0, to 1, to 3, to 6, and finally to 10. In most cases, as shown on the right, a loop is used to repeat the summation algorithm.

Figure 5.8 presents a Java console application named `TotalingLoop` that demonstrates the use of the totaling algorithm and the calculation of an average. A set of sample inputs and the program's corresponding outputs is given at the bottom of the figure. The monetary outputs produced by lines 15–16 and lines 30–34 are formatted as U.S. currency using an object in the `NumberFormat` class declared on line 13. This class and its methods used to produce this currency formatting will be discussed in the next section.

The program accepts a given number of input deposits and uses the totaling algorithm (line 25) to calculate their total. The number of deposits to be processed is input on line 17, then the parsed value is used in the Boolean condition of the `for` statement that begins on line 21 to process that number of deposits.

Before the loop begins, the totaling variable, `total`, is initialized to zero (line 20). During each iteration of the loop that begins on line 21, a new deposit is input and parsed. Then, the total algorithm is used on line 25 to add the new deposit to the total of the previously input deposits. After all the deposits are processed, the new balance (line 27) and the average deposit (line 28) are calculated. Lines 15–16 output the beginning balance, and lines 30–34 output the total of the deposits, the average deposit, and the new balance to the system console.

```
1    import javax.swing.JOptionPane;
2    import java.text.NumberFormat;
3    import java.util.Locale;
4
5    public class TotalingLoop
6    {
7      public static void main(String[] args)
8      {
9        double balance = 1000.24;
10       int numOfDeposits;
11       double deposit, total, newBalance, averageDeposit;
12       String input;
13       NumberFormat us = NumberFormat.getCurrencyInstance(Locale.US);
14
15       System.out.println("Your beginning balance is: " +
16                          us.format(balance));
17       input = JOptionPane.showInputDialog("How Many Deposits?");
18       numOfDeposits = Integer.parseInt(input);
19
20       total = 0.0;
21       for(int i = 1; i <= numOfDeposits;  i++)
22       {
23         input = JOptionPane.showInputDialog("Enter a deposit");
24         deposit = Double.parseDouble(input);
25         total = total + deposit;
26       }
27       balance = balance + total;
28       averageDeposit = total / numOfDeposits;
29
30       System.out.println("The total of the " + numOfDeposits +
31                          " deposits is " + us.format(total));
32       System.out.println("The average deposit was: " +
33                          us.format(averageDeposit));
34       System.out.println("Your new balance is: " + us.format(balance));
35     }
36   }
```

```
Inputs
3
20.10
30.20
40.30

Outputs
Your beginning balance is: $1,000.24
The total of the 3 deposits is $90.60
The average deposit was: $30.20
Your new balance is: $1,090.84
```

Figure 5.8
The application **`TotalingLoops`** and typical inputs and outputs.

5.3 FORMATTING NUMERIC OUTPUT: A SECOND PASS

The use of the `DecimalFormat` class to improve the readability of numeric outputs was briefly discussed in Chapter 2 (Section 2.10). In this section, we will expand that discussion and also discuss the use of the `NumberFormat` and `Locale` classes that were used to format the currency outputs produced by the program shown in Figure 5.8. We will begin with an introduction to the techniques used to format numeric output as currency.

NOTE *All numeric formatting rounds up the fractional part of a numeric value, and all of the digits in the integer portion of the numeric value are always included in the formatted version of the number.*

5.3.1 Currency Formatting

The monetary outputs produced by the program shown in Figure 5.8 are formatted as United States currency. There is a leading dollar sign and a decimal point separating the dollar amount from the cents, which are displayed as two rounded digits to the right of the decimal point. In addition, a comma is used as a thousands separator in the dollar amount. Had the output been negative, it would have been enclosed in parentheses. All of this formatting conforms to the way U.S. currency is displayed within the world of finance and makes the units of the output recognizable as dollars and cents.

Two methods in the `NumberFormat` class, `getCurrencyInstance` and `format`, and constants defined in the class `Locale` can be used to format numeric values as currency. The constants in the class `Locale` are used to specify which of the world's currencies to use in the formatting.

The `NumberFormat` class's method `getCurrencyInstance` is used on line 12 of Figure 5.8 to create a currency formatting object named `us`. The method accepts one argument, which is normally one of the predefined static constants in the `Locale` class. As the name of the class implies, this argument specifies the locale of the format that will be associated with the formatting object. Line 13 uses the constant `Locale.US`, to specify that the currency formatting associated with the object `us` will be United States currency: dollars and cents.

The object `us` is then used on lines 15-16 and lines 30-33 to invoke the `NumberFormat` class's `format` method. This method converts the numeric value passed to it to a string using the formatting associated with the object that invoked it. As a result, the four numeric outputs produced by the program are formatted as U.S. currency.

By changing the argument passed to the method `getCurrencyInstance` on line 13, the numeric output produced by the program could be made to conform to other currency formats used in the financial world. For example, the following code fragment produces the two outputs, which are formatted as pounds (the United Kingdom's currency) and euros (the European Union currency), respectively. The output of the code fragment is given below the code.

```
double price = 1234567.889;
NumberFormat uk = NumberFormat.getCurrencyInstance(Locale.UK);
NumberFormat france = NumberFormat.getCurrencyInstance(Locale.FRANCE);

   System.out.println(uk.format(price));
   System.out.println(france.format(price));
```

Output:

```
£1,234,567.89
1 234 567,89 €
```

NOTE *The formatting performed does not take into account monetary exchange rates.*

The Default Locale

The `getCurrencyInstance` method invoked on line 13 of Figure 5.8 is overloaded. There are two version of it: a one-parameter version that was invoked on line 13 and a no-parameter version. The no-parameter version of the method can be used to format numeric currency in the default locale of the computer's operating system. Assuming the default locale of the operating system was Italy, the following code fragment would produce two identical lines of output formatted as euros (Italy's currency).

```
double price = 1435.2;
NumberFormat italy = NumberFormat.getCurrencyInstance(Locale.ITALY);
NumberFormat osDefault = NumberFormat.getCurrencyInstance();

System.out.println(italy.format(price));
System.out.println(osDefault.format(price));
```

5.3.2 The DecimalFormat Class: A Second Look

The methods in the DecimalFormat class, which were briefly discussed in Section 2.10 of Chapter 2, can also be used to format numeric output. Normally, these methods are used when the numeric value is not currency.

Like the NumberFormat class, the DecimalFormat class contains a nonstatic method named format that returns a string containing the formatted version of the numeric value sent to it as an argument. This method is invoked with a DecimalFormat object that can be declared using the class's one-parameter constructor. The constructor is passed a string argument, called the *formatting string*, which contains the formatting information. In Section 2.10, we used the formatting string argument "#,###.##" to produce an output that contained a comma every three digits to the left of the decimal point and to format real numbers (nonintegers) with a maximum of two digits of precision.

Other characters can be used in the formatting string to produce other forms of numeric output formatting. The pound signs (#) to left and right of the decimal point can be replaced with zeros, which are used to format the numeric value with leading and trailing zeros. In addition, a percent sign (%) can be added to the end of the formatting string. The percent sign is used to format the numeric value as a percentage. For example, the value 0.237 would be formatted as 23.7%, assuming one digit of precision was specified in the formatting string. Numeric values can also be formatted in scientific notation.

Regardless of the characters used in the formatting string, the fractional part of a numeric value is always rounded up and all of the digits in the integer portion of the numeric value or a leading zero are always included in the formatted value unless scientific notation is being used.

Leading and Trailing Zeros

Inserting zeros into the formatting string adjacent to the decimal point will add leading or trailing zeros to the formatted value. For example, when the numeric value being formatted does not have an integer part (e.g., .254), inserting a zero to the left of the decimal point in the formatting string will format the value as 0.254. Adding a zero the right of the decimal point will format the value 167 as 167.0. If the number does have a fractional or integer part, then the digit adjacent to the decimal point in the numeric value always appears in the formatted value.

The code fragment below formats numeric values with one leading zero and two trailing zeros and produces the output shown below the code. The third output value is rounded to the specified two digits of precision.

```
double n1 = .2;
double n2 = 167.0
double n3 = 1.4672
DecimalFormat ltz = new DecimalFormat("#,##0.00");
System.out.print(ltz.format(n1) + " " + ltz.format(n2) + " " +
                 ltz.format(n3));
```

Output:
```
0.20 167.00 1.47
```

Percentages

A formatting string that ends with a percent sign is used to format a numeric value as a percentage. The value will be multiplied by 100, and a percent sign will be added to the right side of the string from the `format` method. For example, the value 0.254 would be formatted as 25.4%. The code fragment below formats numeric values as percentages with one leading zero and one trailing zero. The output it produces is shown below the code.

```
double n1 = 0.002;
double n2 = 0.16 DecimalFormat pct = new DecimalFormat("#,##0.0%");
System.out.println(pct.format(n1) + " " + pct.format(n2));
```

Output:

```
0.2% 16.0%
```

Scientific Notation

Scientific notation is a formatting of a numeric value into a mantissa followed by an exponent. Usually, the mantissa and the exponent are separated by the letter `E`. The mantissa contains the digits of the numeric value with its decimal point shifted left or right. To determine the numeric value, the mantissa is multiplied by 10 raised to the value of the exponent. For example, `23.971E2` represents the numeric value `2,397.1`.

A formatting string that ends with the character `E` followed by the number of leading zeros to be displayed in the exponent is used to format a numeric value in scientific notation. At least one zero must be included after the letter `E` in the formatting string. The formatted value will contain the mantissa and the exponent separated by the letter `E`. The mantissa is formatted using the portion of the formatting string to the left of the letter `E`, which should contain only zeroes and a decimal point.

The code fragment below formats numeric values in scientific notation with the mantissa shown with one digit to the left of the decimal. The output it produces is shown below the code.

```
double n1 = 0.00000215;
double n2 = 16123067533.1
DecimalFormat sn = new DecimalFormat("0.0000E0");
System.out.println( sn.format(n1) + " " + sn.format(n2));
```

Output:

```
2.1500E-6 1.6123E10
```

NOTE *All digits of the exponent will always be included in the scientific formatted version of a numeric value.*

Table 5.1 summarizes the characters used in the `DecimalFormat` class's format string and the formatting they produce. All digits in the integer portion of numeric value will always be included in the formatted version of the numeric unless scientific notation is being used. All digits of the exponent are always displayed when using scientific notation.

Table 5.1
The **`DecimalFormat`** Class's Formatting Characters and Their Meaning

Character	Formatting Produced	Formatting String Example
.	Output a decimal point in this position	
#	Output a digit in this position if it exists	`"#.#"`
0	Output a digit in this position if it exists, else a zero	`"0.00"` One leading zero, two digits of precision
,	Output a comma separator in this position, as necessary	`"#,##0.00"` Comma separator every three digits (with one leading zero and two digits of precision)
%	Output a numeric value as a percentage (Multiply the numeric by 100 and add a percent sign to its right)	`"0.000%"` Convert numeric to a percentage followed by a percent sign (with one leading zero, three digits of precision)
E	Output the numeric value in scientific notation	`"0.0000E0"` Mantissa formatted with 5 digits x.xxxxEx All digits of the exponent are displayed

Figure 5.9 illustrates the use of the methods in the `DecimalFormat` class to format numeric outputs. The program produces four groupings of outputs, which are shown in Figure 5.10. Each grouping outputs the same three numbers: n1, n2, and n3 (lines 15–33) using different formatting strings, which are defined on lines 7–10.

The first three output groupings use comma separators every three digits. The second and third groupings also use leading and trailing zeros, with the outputs in the third grouping displayed as percentages. The fourth output grouping uses scientific notation.

The first numeric output in the first grouping (0.006) has been truncated because its formatting string (line 7) only contains three digits of precision. It contains a leading zero because all numeric values contain a leading zero unless scientific notation is being used. The last two outputs in the first grouping do not contain trailing zeros because the pound sign (#) was used on line 7 to specify their precision.

The outputs in the second grouping contain trailing zeros because a zero was used in their formatting string to specify their precision (line 8). Finally, the exponent (11) in the second numeric output of the last grouping contains two digits even though its formatting string specifies one digit of precision. All the digits of an exponent are always displayed, regardless of the number of zeros used to specify the leading zeros of the exponent.

```
import java.text.DecimalFormat;

public class DecimalFormatClass
{
   public static void main(String[] args)
   {
      DecimalFormat cs = new DecimalFormat("#,###.###"); //commas
      DecimalFormat ltz = new DecimalFormat("#,##0.000"); //zeros
      DecimalFormat pct = new DecimalFormat("#,##0.00%"); //percentages
      DecimalFormat sn = new DecimalFormat("0.0000E0"); //scientific
      double n1 = 0.0062;
      double n2 = 161234563468.5;
      double n3 = 1.530;

      System.out.println("Comma-separators");
      System.out.println(cs.format(n1));
      System.out.println(cs.format(n2));
      System.out.println(cs.format(n3));

      System.out.println("\nLeading & Trailing Zeros, & Commas");
      System.out.println(ltz.format(n1));
      System.out.println(ltz.format(n2));
      System.out.println(ltz.format(n3));

      System.out.println("\nPercentages");
      System.out.println(pct.format(n1));
      System.out.println(pct.format(n2));
      System.out.println(pct.format(n3));

      System.out.println("\nScientific Notation");
      System.out.println(sn.format(n1));
      System.out.println(sn.format(n2));
      System.out.println(sn.format(n3));
   }
}
```

Figure 5.9
The application **DecimalFormatClass**.

```
Comma separators
0.006
161,234,563,468.5
1.53

Leading & Trailing Zeros &Commas
0.006
161,234,563,468.500
1.530

Percentages
0.62%
16,123,456,346,850.00%
153.00%

Scientific Notation
6.2000E-3
1.6123E11
1.5300E0
```

Figure 5.10
The output produced by the application **`DecimalFormatClass`**.

5.4 NESTING FOR LOOPS

As we have learned, loops can be used to repeat a statement or a group of statements contained inside a statement block. When the statement block contains a loop, we say that the loop that is inside the statement block is *nested* inside the other loop. The loop in the statement block is called the *inner* loop because it is inside the other loop, which is referred to as the *outer* loop. The loop in the statement block can be thought of as an egg inside the nest formed by the outer loop. Consider the following code fragment that computes the average of a runner's ten qualifying race times:

```
total = 0;
for(int i = 1; i<=10;  i++)
{
  input = JOptionPane.showInputDialog("Enter a race time");
  aRaceTime = Double.parseDouble(input);
  total = total + aRaceTime;
}
System.out.println(" Your average time is " + total / 10);
```

This code could be used to process 100 runners by nesting it inside an outer loop that executes 100 times.

```
for(int j = 1; j<=100; j++) //each runner (the outer loop)
{
```

```
  total = 0;
  for(int i = 1; i<=10;  i++) //each race (the inner loop)
  {
    input = JOptionPane.showInputDialog("Enter a race time");
    aRaceTime = Double.parseDouble(input);
    total = total + aRaceTime;
  }
  System.out.println(" Your average time is + total / 10);
}
```

As is the case with nested decision statements, there is no limit on how many loops can be nested inside of other loops. The following code fragment processes the 10 qualifying times for 100 racers in 5 states using two levels of nesting.

```
for(int k = 1; k<=5; k++) //each state
{
  for(int j = 1; j<=100; j++) //each runner
  {
    total = 0;
    for(int i = 1; i<=10;  i++) //each race
    {
      input = JOptionPane.showInputDialog("Enter a race time");
      aRaceTime = Double.parseDouble(input);
      total = total + aRaceTime;
    }
    System.out.println("Your average time is + total /10);
  }
}
```

The indentation used in the above code fragment is considered good programming practice because it makes the use of nested loops, and the nesting levels, obvious to anyone reading the code. It can be quickly determined that the three statements in the innermost loop will execute 5,000 times (= 5 * 100 * 10). When using nested loops, it is also good programming practice to progressively develop the code from the inside of the nest outward. The innermost loop (e.g., one that processes 10 races) is coded first, tested, and corrected. Then, this loop is enclosed in a loop (e.g., one that processes 100 runners), and this nested structure is again tested. The process continues until the outermost loop (e.g., one that processes 5 states) is complete.

Figure 5.11 contains a graphics application that illustrates the use of nested `for` loops to draw the checkerboard shown in Figure 5.12, and a second set of nested `for` loops to draw three rows of red checkers on the board. A significant portion of the code is the same as the code shown in Figure 5.6 that drew *one* row of a checkerboard containing red checkers.

Lines 31 to 49 contain the first set of nested `for` loops used to draw the eight rows of the checkerboard. The inner loop, that begins on Lines 33 and ends on Line 42, is same code used on Lines 29 to 38 of Figure 5.6 to draw one row of a checkerboard. This loop is now nested inside an outer loop that begins on line 31, which executes eight times. With each pass through the outer loop, the inner loop draws another row of the board. To prevent the rows from being drawn on top of each other, line 43 increases the y location of the next row of boxes to be drawn in the inner loop by the

```
import edu.sjcny.gpv1.*;
import java.awt.Color;
import java.awt.Graphics;

public class CheckerBoard extends DrawableAdapter
{
  static CheckerBoard ge = new CheckerBoard ();
  static GameBoard gb = new GameBoard(ge, "Nested For loops");

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    int xBox = 12;
    int yBox = 50;
    int boxWidth = 60;
    int boxHeight = 53;
    int firstCheckerCol = 1;
    int checkerX = 20;
    int checkerY = 55;
    Color firstColor = Color.BLACK;
    Color secondColor = Color.RED;
    Color temp;

    gb.setBackground(Color.LIGHT_GRAY);

    //Draw the checker board boxes
    for(int row = 1; row <= 8; row++) //each row
    {
       for(int col = 1; col <=8; col++) //each column
       {
         g. setColor(firstColor);
         if(col % 2 == 0)
         {
           g. setColor(secondColor);
         }
         g.fillRect(xBox, yBox, boxWidth, boxHeight);
         xBox = xBox + boxWidth;
       }
       yBox = yBox + boxHeight;
       xBox = 12;

       temp = firstColor; //swap the box colors
       firstColor = secondColor;
       secondColor = temp;
    }
```

```
50
51       //Draw the red checkers
52       for(int row = 1; row <= 3; row++) //first three rows
53       {
54         if(row % 2 == 0) //an even numbered row
55         {
56           checkerX = checkerX + boxWidth;
57           firstCheckerCol = 2;
58         }
59         g.setColor(Color.RED);
60         for(int col = firstCheckerCol; col <=8; col= col + 2)
61         { //red checker locations
62           checkerX = 20 + (col -1) * boxWidth;
63           g.fillOval( checkerX, checkerY, 40, 40);
64         }
65         checkerY = checkerY + boxHeight;
66         checkerX = 20;
67         firstCheckerCol = 1;
68       }
69     }
70 }
```

Figure 5.11
The graphical application **`CheckerBoard`**.

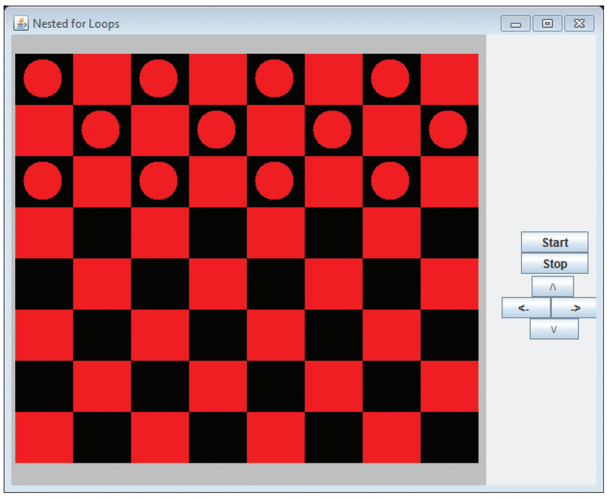


Figure 5.12
The output produced by the application **`CheckerBoard`**.

height of the boxes, and line 44 resets the x location of the each row's first box to 12. Before the outer loop ends, lines 46–48 swap the colors of the odd and even column boxes. This will make the colors of the boxes to be drawn in each column of the next row different from the color of the boxes in the row above them.

The inner loop that begins on lines 60 and ends on line 64 is the same code used on lines 42–46 of Figure 5.6 to draw one row of red checkers. This loop is now nested inside an outer loop that begins on line 31, which executes three times. With each pass through the outer loop, the inner loop draws the next row of `checkers`. After a row is drawn, the y location of the next row of checkers is calculated and assigned to the variable `checkerY` on line 65. The value stored in this variable is increased by the height of the boxes then used during the next iteration of the inner loop (line 63) to draw a row of checkers.

Line 66 reinitializes the x location of the first checker in a row to 20. The `if` statement on line 54 decides when the row number is even. Because only the odd-numbered rows (rows 1 and 3) should have a checker in the first column of the board (at x = 20), `checkerX` is increased by the width of a checkerboard box (line 56) when the row number is even. Then, line 57 sets the variable `firstCheckerCol`, used on line 60 as the column number of a row's first checker, to 2 (line 57)

5.5 THE `WHILE` STATEMENT

Many applications require that a sequence of statements be repeated until a Boolean condition becomes `false`, rather than repeating until the statements have been executed a given or known number of times. For example, a program might continue to ask for a password until the correct password is entered. When this is the case, the `while` or the `do-while` statements are normally used to code the loop that repeats the statements. If the statements should be executed at least once, the `do-while` statement is used. Otherwise, the `while` statement is used. In this section, the `while` statement will be discussed, and the `do-while` statement will be discussed in Section 5.6.

5.5.1 Syntax of the `while` Statement

The generalized syntax of the `while` statement and its execution path are shown on the left and right sides of Figure 5.13, respectively:

```
while  (Boolean expression)
{
   //statement(s) to be repeated
}
```

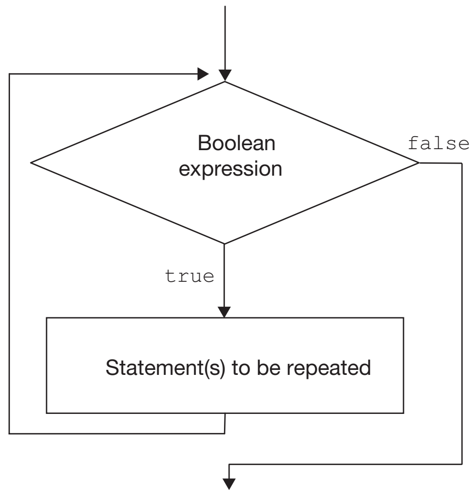


Figure 5.13
The syntax and execution path of the **`while`** statement.

The statement begins with the keyword `while` followed by a Boolean expression enclosed in parentheses, which is followed by a code block containing the statements to be repeated. While it is the case that a single statement to be repeated need not be coded inside a statement block, as discussed in Section 5.2.1, it is good programming practice to do so.

As shown on the right side of Figure 5.13, the statements in the block will be repeated as long as the Boolean condition is `true`. If, when the statement begins execution, the Boolean condition is `false`, the statement block will not be executed.

The following code fragment outputs the square root of 1.2, 2.3, 3.4, and 4.5:

```
double n = 1.2;
while(n != 5.6)
{
```

```
  System.out.println(Math.sqrt(n));
  n = n + 1.1;
}
```

The most common errors made when coding the `while` statement are:

- placing a semicolon after the close parenthesis
- neglecting to code the open and close brace around the statements when more than one statement is to be repeated

When a semicolon is coded after the close parentheses, the statement is syntactically correct, however, none of the statements that would have normally formed the loop body are considered to be part of (inside) the loop. They default to sequential execution, and each statement executes once. When braces are not coded, the statement is also syntactically correct, however, the first statement after the `while` statement is the only statement considered to be part of the loop. Regardless of the indentation used, it is the only statement repeated. When coding a `while` loop, it is good programming practice to code the following fragment and then add the Boolean condition and the statements to be repeated:

```
while()
{

}
```

A common logic error made when coding the `while` statement is that the statements inside the statement block, during some repetition of the loop, do not change the Boolean expression to `false`. The statement, or statements, intended to do that were either incorrectly coded or were not included in the loop's statement block. In either case, once the loop begins, it never ends, and the loop is said to be an *infinite* loop. The following code fragment is an infinite loop because the statement that increments `n` is not part of the loop's statement block. On each iteration through the loop, `n`'s value remains 1.2, and the Boolean condition never becomes `false`.

```
double n = 1.2;
while(n != 5.6)
{
  System.out.println(Math.sqrt(n));
}
n = n + 1.1;
```

Infinite loops can also occur when the loop's Boolean expression is improperly coded, as is the case in the following code fragment. The variable `n` assumes the values 1.2, 2.3, 3.4, 4.5, 5.6 …, but never the value 5.5.

```
double n = 1.2;
while(n != 5.5) //n never becomes 5.5
{
  System.out.println(Math.sqrt(n));
  n = n + 1.1;
}
```

5.5.2 Sentinel Loops

Many applications require that a sequence of instructions be repeated until a signal to stop is detected. The signal is referred to as a sentinel value, and a loop that ends when it detects a sentinel value is called a *sentinel* loop.

Sentinel loops are commonly used to process a set of input data, and the sentinel value is chosen to be a specific value of the input data. The value selected must be a value that would never occur in that data set (for example, a student grade of -1). As another example, you might want to continue to process bank deposits until a negative deposit is entered, or when data is being read from a disk file to continue to process data until an End of File (EOF) marker is encountered. Although `for` loops that contain `break` statements could be used to code sentinel loops, they are more easily coded using the `while` and `do-while` statements.

Often, the use of sentinel loops in our programs makes them easier to use. Imagine you are a data-entry person using the program shown in Figure 5.8 to process a group of input deposits. When the program is launched, you are asked for the number of deposits. If the first item on the list of deposits you were given to enter was the number of deposits, the automatic counting performed by the `for` loop used in the program would be perfect for the application.

However, if the number of deposits was not included in the list of deposits, then before you used the application, you would have to count the number of deposits. Not only would this be time consuming when the list of deposits was long, but if you miscounted the number of deposits, the program would either terminate before all the deposits were entered (because your count was too low) or ask you to enter a deposit that did not exist (because your count was too high). Generally speaking, when the number of data items to be processed is not easily determined, sentinel loops make our programs much easier to use.

The following code fragment is a template for a sentinel loop that uses a `while` statement to process a set of inputs. Each input is read into a variable called the sentinel variable.

```
//obtain the first input into the sentinel variable
while( //the input is not the sentinel value )
{
    //statement(s) to perform the loop's processing
    //obtain the next input into the sentinel variable
}
```

The template begins with a statement to accept the first input and the same statement, which accepts all subsequent inputs, is also coded as the last statement in the `while` statement's code block. The other statements in the statement block perform the processing of each input. This placement of the input statements in a sentinel loop prevents the processing of the sentinel value, even if it is the first input (i.e., the data set is empty).

The three most common errors made when coding a sentinel loop are:

- neglecting to code the statement to accept the first sentinel variable input before the `while` statement

- neglecting to code the statements to accept all subsequent inputs of the sentinel variable at the end of the loop's code block
- coding the statements to accept all subsequent inputs of the sentinel variable inside the loop's code block before the processing statements

The first error results in the loop processing the default value of the sentinel variable, or if the default value is the sentinel value, the loop does not execute at all. The second error results in an infinite loop because once the loop is entered, the sentinel variable is not changed. When the third error is made, the first input is not processed.

The application `SentinelWhileLoop` presented in Figure 5.14 demonstrates the use of the code template. It is a sentinel loop version of the program shown in Figure 5.8 that totals and averages a set of input deposits. Typical inputs and outputs are shown at the bottom of Figure 5.14. As previously discussed, this version would be preferred if the number of deposits was not the first data item.

Following the format of the `while` loop sentinel template, line 18 accepts the initial value of the sentinel variable `input`. The Boolean expression (on line 19) of the `while` statement uses this variable to decide if the statement's code block should be executed. If anything other than the sentinel value (-1) has been input, an execution of the loop's statement block is performed.

Because an average is to be calculated and the user is no longer required to enter the number of deposits, the counting algorithm is used on line 23 to count the number of deposits processed. The counting variable, `numOfDeposits`, is initialized to zero on line 15. Consistent with the `while` sentinel loop template, the last line of the loop's code block (line 24) accepts the next input value and stores it in the sentinel variable `input`.

```
1    import javax.swing.JOptionPane;
2    import java.text.NumberFormat;
3
4    public class SentinelWhileLoop
5    {
6      public static void main(String[] args)
7      {
8        double balance = 1000.24;
9        int numOfDeposits;
10       double deposit, total, newBalance, averageDeposit;
11       String input;
12       NumberFormat us = NumberFormat.getCurrencyInstance();
13
14       System.out.println("Your beginning balance was:
                                "+ us.format(balance));
15       numOfDeposits = 0;
16       total = 0.0;
17
18       input = JOptionPane.showInputDialog("Enter a deposit, -1 to end");
19       while( !input.equals("-1") ) //input is not "-1"
```

```
      {
        deposit = Double.parseDouble(input);
        total = total + deposit;
        numOfDeposits++;
        input = JOptionPane.showInputDialog("Enter a deposit, -1 to end");
      }

      balance = balance + total;
      averageDeposit = total / numOfDeposits;

      System.out.println("The total of the " + numOfDeposits +
                         " deposits is "+ us.format(total));
      System.out.println("The average deposit was: " +
                          us.format(averageDeposit));
      System.out.println("Your new balance is: " + us.format(balance));
    }
  }
```

Inputs
20.10
30.20
40.30
-1

Outputs
Your beginning balance was: $1,000.24
The total of the 3 deposits is $90.60
The average deposit was: $30.20
Your new balance is: $1,090.84

Figure 5.14
The console application **SentinelWhileLoop** and the output it produces.

Another commonly used form of a `while` sentinel loop parses the input after each input is accepted. This adaptation of the `while` loop sentinel template in the program shown in Figure 5.14 would be coded as:

```
input = JOptionPane.showInputDialog("Enter a deposit, -1 to end");
deposit = Double.parseDouble(input);
while (!deposit == -1.0)
{
  //statement(s) to perform the loop's processing, less the parsing
  input = JOptionPane.showInputDialog("Enter a deposit, -1 to end");
  deposit = Double.parseDouble(input);
}
```

5.5.3 Detecting an End Of File

Often, large data sets are stored in disk files, and the programs that process them read the data from the disk file until a sentinel value is detected. Because the sentinel value is chosen to be a value outside the range of the data set's values, most sentinel values vary from one application to another. In the case of a disk-based data file, there is one sentinel value that works for all data sets: the End of File (EOF) marker that is placed at the end of each file.

In Section 4.8.1 of Chapter 4, we used the methods in the `Scanner` class to read data from a disk text file. The `Scanner` class also contains a method named `hasNext` that can be used to detect the EOF marker in a file. The method has no parameters and returns `false` when the EOF marker is encountered. The following code fragment uses a sentinel loop to read all the integer data values from the disk text file `data.txt` stored on the root of the C drive and outputs the values to the system console. The value returned from the method `hasNext` is stored in the sentinel-variable `notEOF`.

```
int dataItem;
File fileObject = new File("c:/data.txt");
Scanner fileIn = new Scanner(fileObject);
boolean notEOF; //the sentinel variable

notEOF = fileIn.hasNext(); //fetch the 1st sentinel variable value
while(notEOF) //more data to process
{
  dataItem = fileIn.nextInt();
  System.out.println(dataItem);
  notEOF = fileIn.hasNext(); //fetch next sentinel value
}
fileIn.close();
```

The following code fragment is a more succinct and more commonly used version of an EOF sentinel loop:

```
int dataItem;
File fileObject = new File("c:/data.txt");
Scanner fileIn = new Scanner(fileObject);

while(fileIn.hasNext()) //more data to process
{
    dataItem = fileIn.nextInt();
    System.out.println(dataItem);
}
fileIn.close();
```

Figure 5.15 presents a modified version of the program shown in Figure 5.14 that processed a group of deposits entered from the keyboard. The new version of the program reads the deposits from a disk file using the file's EOF marker as a sentinel value. A set of file inputs and the corresponding program outputs are given at the bottom of Figure 5.15.

The instructions in Figure 5.14 that accept input from the keyboard have been removed from the program. Lines 1 and 2 of Figure 5.15 have been added to access the `Scanner` and `File` classes. In the interest of brevity, exceptions that could be generated during the disk file input performed by the program are not processed by the modified program. Rather, a `throws` clause has been added to the end of line 7.

Line 21 begins a sentinel `while` loop that ends when an EOF marker is detected. Inside the loop, line 23 reads and parses the next deposit using the `Scanner` object created by lines 13 and 14. Line 35 closes the disk file after the console output is performed.

```
1    import java.util.Scanner;
2    import java.io.*;
3    import java.text.NumberFormat;
4
5    public class EndOfFile
6    {
7      public static void main(String[] args) throws IOException
8      {
9        double balance = 1000.24;
10       int numOfDeposits;
11       double deposit, total, newBalance, averageDeposit;
12       NumberFormat us = NumberFormat.getCurrencyInstance();
13       File fileObject = new File("c:/data.txt");
14       Scanner fileIn = new Scanner(fileObject);
15
16       numOfDeposits = 0;
17       total = 0.0;
18
19       System.out.println("Your beginning balance is: " +
20                          us.format(balance));
21       while(fileIn.hasNext()) //more data to process
22       {
23         deposit = fileIn.nextDouble();
24         total = total + deposit;
25         numOfDeposits++;
26       }
27       balance = balance + total;
28       averageDeposit = total / numOfDeposits;
29
30       System.out.println("The total of the " + numOfDeposits +
31                         " deposits is " + us.format(total));
32       System.out.println("The average deposit was: " +
33                          us.format(averageDeposit));
34       System.out.println("Your new balance is: " + us.format(balance));
35       fileIn.close();
36     }
37   }
```

Disk File Inputs
20.10
30.20
40.30

Outputs
Your beginning balance was: $1,000.24
The total of the 3 deposits is $90.60
The average deposit was: $30.20
Your new balance is: $1,090.84

Figure 5.15
The application **`EndOfFile`**, a set of file inputs, and corresponding outputs.

An alternate approach to using the `Scanner` class's `hasNext` method to detect the end of a file is to write the number of data items into the file as the file's first value. If the data file processed by the program shown in Figure 5.15 had been written this way, lines 21-26 would be coded as shown below:

```
21      numOfDeposits = fileIn.nextInt();  //read the number of data items
22       for(int i = 1; i <= numOfDeposits; i++) //each data item
23       {
24         deposit = fileIn.nextDouble();
25         total = total + deposit;
26       }
```

Most often, the use of the `hasNext` method to detect the end of the file is the better approach because if new data were added to end of the file, updating the number of data items at the beginning of the file would require reading, deleting, and rewriting the entire file. This is a time-consuming process.

5.6 THE DO-WHILE STATEMENT

As previously discussed, when a loop in an application is to execute a known number of times, the `for` statement is best suited for the application. When the number of times to execute the loop is not known, either the `while` or the `do-while` statements are preferred. Of these two statements, the `do-while` statement is the better alternative when the loop's statements should be executed *at least once*. For example, the code to check a password is normally coded inside a `do-while` loop because the password has to be entered at least once.

5.6.1 Syntax of the do-while Statement

The generalized syntax of the `do-while` statement and its execution path are shown on the left and right sides of Figure 5.16, respectively.

```
do
{
  //statement(s) to be repeated

}while (Boolean expression);
```

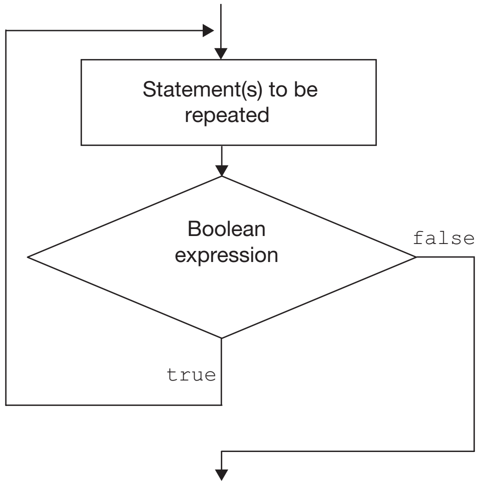


Figure 5.16
The generalized syntax of the **do-while** statement.

The statement begins with the keyword **do** followed by the loop's statement block. The keyword **while** and the statement's Boolean expression enclosed in parentheses are coded after the statement block's close brace. It is good programing style to code the keyword **while** and the Boolean expression on the same line as the close brace because it improves the readability of the statement. The `do-while` statement ends with a semicolon coded after the close parentheses that terminates the Boolean expression.

As shown on the right side of Figure 5.16, the loop's statement(s) will be executed at least once because they are executed before the Boolean condition is tested. After they execute, the Boolean expression is tested, and the statements are repeated until the Boolean condition becomes `false`. The `do-while` loop is called a *post-test* loop because the test to terminate the loop is performed after the loop's statement block has executed at least once.

The most common syntactical errors made when coding the `do-while` statement are:

- placing a semicolon after the keyword **do**
- neglecting to include a semicolon after the Boolean expression
- neglecting to code the open and close braces around the statements when more than one statement is to be repeated

All of these coding errors are syntax errors and are detected and reported by the Java translator. When coding a `do-while` loop, it is good programming practice to code the following code fragment and then add the statements to be repeated and the Boolean condition, even if only one statement is to be repeated.

```
do
{

}while( );
```

The most common logic error made when coding this statement is that the statements inside the statement block, during some repetition of the loop, do *not* change the Boolean expression to `false`. In this case, once the loop begins, it never ends and is said to be an infinite loop. For example, the following code sequence is an infinite loop because the loop's statement does not change the string variable `password`, which is used in the Boolean condition. The user-entered password is mistakenly stored in the string object `pw` leaving the string `password` unchanged each time through the loop.

```
String password = "";
String pw;
do
{
   pw = JOptionPane.showInputDialog("Enter The Password");
}while(!password.equals("Mercury"));
```

The following code fragment is the correct coding of a `do-while` statement that verifies the entry of the correct password, *Mercury*.

```
String password = "";
do
{
   password = JOptionPane.showInputDialog("Enter The Password");
}while(!password.equals("Mercury"));
```

Infinite loops can also occur when the loop's Boolean expression is improperly coded. In the code fragment below, the logical operator NOT(`!`) has been left out of the Boolean expression, and any password other than the correct password, *Mercury*, is accepted.

```
String password = "";
do
{
   password = JOptionPane.showInputDialog("Enter The Password");
}while(password.equals("Mercury"));
```

5.7 THE BREAK AND CONTINUE STATEMENTS

The `break` and `continue` statements are used inside a `for`, `while`, or `do-while` loop's coded block to alter the loop's execution path. Just as a `break` statement terminates the execution of a `switch` statement, a `break` statement may also be used to terminate a loop. Execution continues with the statement that follows the loop.

NOTE *When a `break` statement is executed inside a loop, the execution of the loop terminates.*

When a `continue` statement is executed in a loop, the *current* iteration of the loop is terminated, and statements that come after it in the loop's body are not executed during that iteration of the loop. Execution continues with the testing of the loop statement's Boolean condition. When a `continue` statement is executed inside a `for` loop, the loop variable is incremented before the Boolean condition is tested.

NOTE *When a `continue` statement is executed inside a loop, the current iteration of the loop terminates.*

To illustrate the use of these two statements, the code fragment below gives a game player three chances to enter the password "*Mars*" to access a game. The `break` statement is used to exit the loop after the message *Look up your password* is output. The `continue` statement is used to skip the message box output and the `break` statement until three incorrect passwords are entered.

```
int count = 1;
String password = "";
do
{
   if(count <= 3)
   {
     password = JOptionPane.showInputDialog("Enter your password");
     count++;
     continue;
   }
   JOptionPane.showMessageDialog(null, "Look up your password");
   break;
}while(!password.equals("Mars"));
```

5.8 WHICH LOOP STATEMENT TO USE

When the number of times to repeat the loop's statements is known, the `for` statement should be used to code the loop. The number of times to repeat the loop could be known at the time the program is written (e.g., the program will always process 100 race times), or it is determined during the program's execution before the loop statement is executed. For example, before entering a group of deposits, the program users are asked to enter the number of deposits they will be making into their bank account. In both of these cases the `for` loop is the preferred loop statement.

When the number of times to execute the loop is not known, either a `while` or a `do-while` loop is preferred to a `for` loop. The `while` statement is the better alternative when there are times (cases) when the loop body should not execute even once. The `do-while` statement is the better alternative when the loop's statements should always be executed at least once. Table 5.2 summarizes the criteria for selecting the best loop statement for a particular application.

Table 5.2
Criteria for Selecting the Best Loop Statement

Is the Number of Times the Loop Will Execute Known?	Loop Statement
Yes	`for`
No, and there are cases when the loop body should *not* execute even once	`while`
No, and the loop body should always execute *at least once*	`do-while`

The boundaries between the use of the three loop statements become somewhat blurred with the use of the counting algorithm inside a `while` loop and the use of a `break` statement inside a `for` loop. For example, to average 100 items, most programmers would use a `for` loop. However, a `while` loop that contains the counting algorithm can also be used, as illustrated in the following code fragment:

```
int count = 1;
double total = 0;
double average;
String sItem;
while (count <= 100)
{
   sItem = JOptionPane.showInputDialog("enter an item");
   total = total + Double.parseDouble(sItem);
   count++;
}
average = total / 100;
System.out.println("The average of the 100 items is: " + average);
```

The `for` loop is preferred for this application because when a `while` loop is used and we neglect to increment the counter (`count++;`) in the loop's body, the loop becomes an infinite loop. If we use a `for` loop and neglect to increment the counter in the first line of the `for` statement, the translator would alert us to the oversight.

Consider a program that totals input items until a -1 is entered or 100 items have been entered. Most programmers would use a `while` loop for this application. However, it can be coded using a `for` loop that contains a `break` statement.

```
double total = 0;
String sItem;

for(int i = 1; i <= 100; i++)
{
   sItem = JOptionPane.showInputDialog("enter an item");
   if(sItem.equals("-1"))
   {
      break;
   }
   total = total + Double.parseDouble(sItem);
}
System.out.println("The total of the items is: " + total);
```

When this `for` loop is used it would appear that the loop will always execute 100 times. However, the loop terminates before 100 iterations when the `break` statement executes. From a code readability point of view, the following `while` loop is the preferred loop statement for this application because the first line of the `while` loop clearly states the two conditions that will end the input loop.

```
int count = 1;
double total = 0;
String sItem;
```

```
sItem = JOptionPane.showInputDialog("enter an item, or -1");
while (!sItem.equals("-1") && count <= 100) //tests both conditions
{
   total = total + Double.parseDouble(sItem);
   count = count++;
   sItem = JOptionPane.showInputDialog("enter an item, or -1");
}
System.out.println("The total of the items is: " + total);
```

5.9 THE RANDOM CLASS

Pseudorandom numbers, their use in computer programs, and the ability to generate them with the `Math` class's `random` method were discussed in Section 2.6.4 of Chapter 2. The methods in the class `Random` can also be used to generate pseudorandom numbers. In fact, these methods do the work of the `Math` class's `random` method in that the method `random` invokes the `Random` class's methods to generate the numbers it returns.

Table 5.3 lists the `Random` class's constructors and some of its methods used to generate pseudorandom numbers. Each time these methods are invoked, they return the next number in a sequence of random numbers. An object in the class `Random` is used to invoke them, which is created using one of the class's constructors.

```
Random randomObject1 = new Random();
Random randomObject2 = new Random(123456);
```

Table 5.3
`Random` Class Methods

Method	Description	Coding Example
`Random()`	Creates a `Random` object based on the seed value time of day	`Random ro = new Random();`
`Random(long seed)`	Creates a `Random` object based on the seed argument sent to it	`Random ro = new Random(675);`
`nextDouble()`	Returns the next pseudorandom real number in the range: 0.0<=randomNumber<1.0	`double rn = ro.nextDouble();`
`nextInt()`	Returns the next pseudorandom integer in the range of the `int` primitive type	`int rn = ro.nextInt();`
`nextInt(int max)`	Returns the next pseudorandom integer in the range zero to one less than max	`int rn = ro.nextInt(20);`

When the one-parameter constructor is used to create the object, the sequence of numbers the methods generate is based on the integer argument sent to the constructor, which is called a *seed* value. When the no-parameter constructor is used to create the object, the sequence of numbers the methods generate is based on the time of day because the seed value defaults to the real-time clock's value expressed in milliseconds. Sequences of numbers generated with objects created using the same seed value will be identical.

The one-parameter constructor is used when it is desirable to generate the same sequence of pseudorandom numbers every time the program is run. Conversely, the no-parameter constructor is used when it is desirable to generate a sequence of pseudorandom numbers that rarely repeats because the program would have to be run at exactly the same time of day to generate the same sequence of numbers.

Like the `Math` class's `random` method, the method `nextDouble` generates and returns a pseudorandom real number (a `double`) in the range: 0.0 <= randomNumber < 1.0. The method can be used to generate a real number in the range: min ≤ randomNumber < max using the following assignment statement (and sample object declaration):

```
Random randomObject2 = new Random(98765);
randomNumber = min + randomObject2.nextDouble() * (max - min);
```

The following code sequence outputs a sequence of ten pseudorandom real numbers in the range 20.0 ≤ randomNumber < 50.0. It would be very unusual for this code to generate the same sequence of numbers during two executions of the program because the sequence's seed value is the time of day in milliseconds. (The `Random` object is created with the no-parameter constructor.) Alternately, the one-parameter constructor could be used to generate a repeatable sequence of numbers.

```
double randomNumber;
double min = 20.0;
double max = 50.0;
Random randomObject2 = new Random(); // time of day seed value

for(int i = 1; i<=10; i++)
{
  randomNumber = min + randomObject2.nextDouble() * (max - min);
  System.out.println(randomNumber);
}
```

As shown in the Table 5.3, there are two versions of the `nextInt` method, which is used to generate and return a random integer (of type `int`). The no-parameter version returns a pseudorandom number over the full range of an `int` type variable (see Table 2.1).

The one-parameter version of the `nextInt` method is used to generate a sequence of integers, each of which are within a specified range. The numbers returned from the method are in the range zero to one less than the argument sent to it. The following code sequence generates a pseudorandom number between zero and nine, inclusive:

```
Random randomObject2 = new Random();
randomNumber = randomObject2.nextInt(10)
```

The use of this method can be generalized. The following code sequence outputs ten random integers in the range three to six, *inclusive*. The initial values of the variables `max` and `min` specify the lowest and highest numbers generated in the sequence. Every time this code is run, the same sequence of numbers is generated (5, 5, 5, 4, 3, 5, 5, 3, 6, 4) because the one-parameter constructor was used to create the `Random` object.

```
int randomNumber;
int min = 3;
int max = 6;
Random randomObject1 = new Random(98765); //repeatable random set

for(int i = 1; i<=10; i++)
{
  randomNumber = min + randomObject1.nextInt(max - min + 1);
  System.out.println(randomNumber); //in the range min to max inclusive
}
```

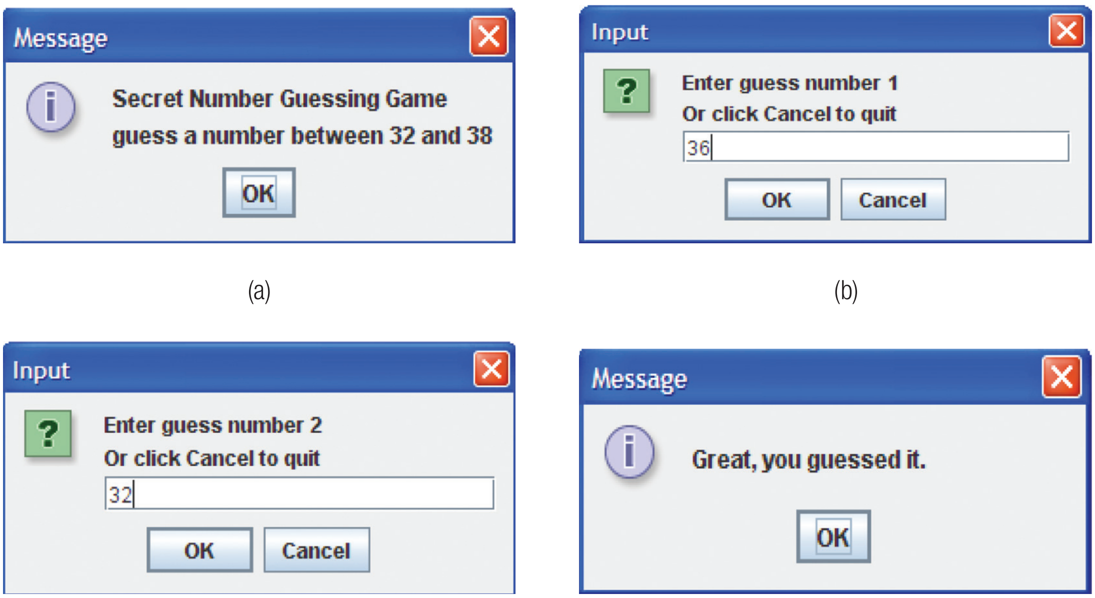
Figure 5.17 shows a number-guessing game program in which the player is asked to guess a number between 32 and 38, inclusive. The inputs and outputs for a correct answer on the second guess are shown in Figure 5.18.

Line 1 imports the `Random` class into the program. Line 8 declares an instance of this class, `randomObject`, which is used to invoke the `nextInt` method on line 15. The use of the no-parameter constructor on line 8 ensures that each time the program is run, there is the possibility that a different pseudorandom number will be generated by line 15. The maximum and minimum values of the pseudorandom numbers used on line 15 are specified on lines 9 and 10.

```
1   import java.util.Random;
2   import javax.swing.*;
3
4   public class RandomClass
5   {
6     public static void main(String[] args)
7     {
8       Random randomObject = new Random(); //time of day seed value
9       int min = 32;
10      int max = 38;
11      int secretNumber;
12      String sGuess;
13      int count = 1;
14
15      secretNumber = min + randomObject.nextInt(max - min + 1);
16      JOptionPane.showMessageDialog(null, "Secret Number Guessing Game" +
17                                    "\nguess a number between " +
18                                      max + " and " + min);
19      do
20      {
```

```
      sGuess = JOptionPane.showInputDialog("Enter a guess " + count +
                                 "\nOr click Cancel to quit");
      count++;
      if(sGuess == null) //Cancel was clicked
      {
        break;
      }
    }while(secretNumber != Integer.parseInt(sGuess));

    if(sGuess == null) //Cancel was clicked
    {
      JOptionPane.showMessageDialog(null, "Secret Number was " +
                                 secretNumber);
    }
    else
    {
      JOptionPane.showMessageDialog(null, "Great, you guessed it.");
    }
  }
}
```

Figure 5.17
The application **`RandomClass`**.



(a) (b) (c) (d)

Figure 5.18
The inputs and corresponding outputs produced by the application **`RandomClass`**.

5.9.1 The SecureRandom Class

As previously mentioned, the random numbers generated by the `Math` class's `random` method and the `Random` class's methods appear to be perfectly random, but in fact they can be predicted. Thus, they are more correctly referred to as pseudorandom numbers. This is usually not a concern when the random number is used to position a game piece at a random location on a game board, but it can be a concern if the random number is used within a security system.

When this is the case, an object in the `SecureRandom` class should be used to generate the random numbers. Its use is similar to the use of the `Random` class except that we invoke the methods described in the last three rows of the Table 5.3 on a `SecureRandom` object to generate the random numbers. To convert the application shown in Figure 5.17 to generate a non-pseudorandom number to be guessed by the user, the variable `randomObject` declared on line 8 of the application is declared to be an instance of the `SecureRandom` class. The remainder of the application is unchanged. The new line 8 of Figure 5.17 is:

```
SecureRandom randomObject = new SecureRandom();
```

Numbers generated by a `SecureRandom` object are not predictable. This added security does not come for free in that there is a decrease in speed performance associated with the use of this class. When security is not a serious issue, a `Random` object should be used to generate the (pseudo) random numbers.

5.10 THE ENHANCED for STATEMENT

The enhanced `for` statement, sometimes referred to as the for-each statement, is an alternate syntax of a `for` loop that is used to fetch *all* of the elements of an array sequentially. During the execution of the loop, the elements of the array cannot be modified, so its use is limited. It can be used to output or total the elements of an array.

The syntax of the statement is given below, in which `anElement` is a variable whose type, `aType`, is always the type of the elements of the array `arrayName`:

```
for( aType anElement: arrayName )
{
  //statement(s) that use the variable anElement
}
```

For example, if the array is an array of references to `String` objects, the statement would be coded as shown below:

```
for( String anElement: arrayName )
{
  //statement(s) that use the variable anElement
}
```

A colon is always coded after the variable `anElement`. If there is only one statement to be executed within the loop, the open and close braces need not be coded, but it is good programming practice to include them.

The number of times the loop executes is always equivalent to the length of the array, in the above case `arrayName.length`, unless the loop contains a break statement. During the execution of the loop, the variable `anElement` assumes the value of each element of the array in ascending order, beginning with the first element (during the first iteration of the loop) and ending with the last element (during the last iteration of the loop). The two loops shown below are equivalent, and both produce the system console output `Nora Ryan Logan`.

```
String anArray = {"Nora", "Ryan", "Logan");

for(String anElement: anArray)
{
  System.out.print (anElement + " ");
}

System.out.println( );

for(int i = 0; i < 3; i++)
{
  System.out.print (anArray[i] + " ")
}
```

Within the loop's body, the variable used in the enhanced `for` statement (e.g., `anElement`), can be used anywhere it is syntactically correct to use a variable of its type. An advantage of the enhanced `for` loop is that it cannot produce an `ArrayIndexOutOfBounds` error because it does not use a loop variable to access the elements of the array. The disadvantage is that the elements of the array cannot be changed inside the loop. We will see a more practical use of the enhanced for statement in Chapter 13, "Generics."

5.11 CHAPTER SUMMARY

Many applications require that the statements in a statement block be repeated, and in this chapter we discussed three ways to perform this repetition: a `for` loop, a `while` loop, and a `do-while` loop. The `for` loop is an automatic counting loop used when the number of times to repeat the statements is known. The `do-while` and `while` loops end when their Boolean condition becomes `false` and they are usually used to detect a sentinel value of the data they are processing. The `do-while` loop is used whenever the loop's block should be executed at least once, and the `while` loop is a more general-purpose loop that can be used in most applications.

The loop control variable of a `for` loop is used to control the number of iterations of the loop. The statement's initialization expression sets its initial value. At the end of each loop iteration, the increment expression executes, which normally changes the value of the loop variable. The `for` and `while` loops are called pretest loops: they test their Boolean condition to continue at the beginning of the loop; the `do-while` loop is a posttest loop, testing its condition to continue at the end of an iteration.

The totaling or summation algorithm is a loop-based algorithm because it sums a set of items by repeatedly adding the value of a new item to an existing subtotal and making the result the new subtotal. Its template is: `total = total + newItem`. Each time through the loop, the value of the variable `newItem` assumes the next value to be totaled, which is often input by the user or from a disk file. The variable total is initialized to zero before the totaling loop begins. The counting algorithm can use used inside a loop's statement block to count the number of times the loop executes and can then be divided into the total the loop calculates to determine the average of the totaled values.

Often, a sentinel value is used to terminate a loop when the number of input values to be processed is unknown. Two Java statements, `break` and `continue`, also enable us to control the number of times all or some of the statements within the body of the loop will be executed. When a `break` is executed within a loop, the loop terminates. The `continue` statement can be used to end the current iteration of the loop and is useful when conditions dictate that the remaining statements in the loop's block should be skipped during the current iteration. When a loop is used to obtain and process an unknown number of inputs from a file, Java's End of File (EOF) character or the `Scanner` class's `hasnext` method can be used as a sentinel to terminate the loop.

Nested loops are used to repeat loop-based algorithms. Examples include averaging 10 grades and repeating this for 20 students or processing a set of race times for 100 runners. Nested loops are particularly useful in creating two-dimensional graphics that are composed of many instances of the same repetitive shape, such as the eight rows of eight squares of a checkerboard.

The `Random` class's `nextInt` and `nextDouble` methods can be used to generate a random integer or real number and, when used inside of a loop, to generate a set of random numbers. The `nextInt` method is easier to use than the `Math` class's `random` method because it returns an integer in the positive range of the `int` type or within a specified range. This makes it ideal for generating random game board pixel locations. In addition, when the methods are invoked using a `Random` object created with the class's one-parameter constructor, they generate a repeatable sequence of pseudorandom numbers. This is particularly useful in applications that require the same starting point every time they are launched and is always used when the random numbers are generated within a graphics call back method.

The methods in the `DecimalFormat` class can be used to insert leading/training zeros and comma separators into numeric output, specify the output's precision, and convert the output to a percentage or display it using scientific notation. The methods in the `NumberFormat` and `Locale` classes are used to produce local dependent currency formatting for use in financial and international applications.

Our knowledge of these statements will be expanded in Chapter 6, which covers the concept of arrays because loops are used to unlock the power of arrays. Also, in the next chapter, we will see how loops can be used with arrays to enable us to input, output, and process large data sets.

Knowledge Exercises

1. True or false:
   a) The body of a `while` loop will always execute at least once.
   b) The `for` loop is an automatic counting loop and should be used where the number of repetitions is known.
   c) A sentinel is a data value that can be used to terminate a loop.
   d) The `do-while` loop will continue until the Boolean expression in the `while` statement becomes `true`.
   e) A `while` loop is a posttest loop.
   f) Checking for the EOF condition can be used to control a loop.
   g) A nested loop is a loop within a loop.
   h) Every `while` loop can be written as a `for` loop without using a break statement.
   i) Every `for` loop can be written as a `while` loop.
   j) The `break` statement ends the current iteration of a loop.
   k) A `for` loop ends when Boolean condition becomes `true`.
   l) The `continue` statement can be coded inside any loop.
   m) Placing a semicolon after the parenthesis in a `while` loop can cause an infinite loop.
   n) The statement block of a do-`while` loop may not be executed.
   o) A `for` loop may be designed to count down by decrementing the control variable.
   p) Loops may be used to validate user input or to give the user another chance to enter a value, such as a password, that was typed incorrectly.

2. Write a loop that outputs the integers from 20 to 100 to the system console and the appropriate term, `odd` or `even`, next to each output value.

3. Write a loop that outputs the sum of the `even` integers from 1 to n, where n is a value input by the user, to a message box.

4. Consider the following code fragment:

```
int i = 10;
int  sum = 0;
while (i <= 100)
{
   sum = sum + i;
   i++;
}
System.out.println("The sum of the integers from 10 to 100 is: " +
                   sum);
```

   a) How many times does this loop execute?
   b) Write an equivalent `for` loop.
   c) Write an equivalent `do-while` loop.

5. Consider the following code fragment:

```
int i = 1;
while (i != 20)
{
   i = i + 2;
}
System.out.println("The value of i is " + i);
```

a) Will this loop terminate? If not, why not?

b) What numbers does it output?

6. Consider the following code fragment:

```
int num = 4;
for (int i = 2; i <= 7; i++)
{ System.out.println( "Number is " + num);
  num = num + i;
}
```

a) What is the value of `num` after the loop has executed twice?

b) How many times will the body of the loop be executed?

c) What value will be output on the fourth time through the loop?

d) What is the value of `num` when the loop ends?

e) What causes this loop to terminate?

7. Consider the following code fragment:

```
int x = 11;
while (x > 0)
{  x = x - 3;
   System.out.println(x);
}
```

a) Give its output

b) Write an equivalent `for` loop

8. Write the code fragment for an input validation loop that asks a user to enter an integer in the range of zero to five, displays an error if the input is out of range, and gives the user an *unlimited* number of chances to enter it correctly.

9. Write the code fragment for an input validation loop that asks a user to enter an integer in the range of zero to five, displays an error if the input is out of range, and gives the user at most three chances to enter it correctly.

10. Explain the difference in the execution paths of a `while` loop and a `do-while` loop.

11. Give a code fragment to produce the following output to the system console every time it is executed:

a) A different set of 20 random integers in the range 0 to 500 using the `Random` class and then using the `SecureRandom` class.

b) The same set of 20 random integers in the range 0 to 500

c) The same set of 20 random integers in the range 7 to 500

d) The same set of 20 random integers in the range `min` to `max`

12. Give the declarations and output statements required to display the value stored in the double variable `balance`, formatted as specified below. Also give the resulting formatted output.

a) US currency

b) One leading and one trailing zero, with comma separators every three digits to the left of the decimal point

c) Scientific notation with four digits of precision

d) Two trailing zeros, comma separators every three digits to the left of the decimal point, and a leading zero only when the balance contains a value that only has a fractional part

e) Spanish currency

Programming Exercises

1. Write a program that uses a `for` loop to calculate and output the product of the integers from n to 1 (n factorial) to a message box. For example, when n = 4, the output would be 24 = 4 * 3 * 2 * 1. The value of n will be input by the user via an input dialog box, and the output should be properly annotated.

2. You have just been hired by the TravelStars agency, and your first assignment is to produce a histogram to graphically represent the ratings that travelers have given to various hotels. Your program will begin by asking the user to enter the number of hotels to be including in the histogram. Then, ask a user to input each hotel's name, the hotel's star rating, an integer between 1 and 10 stars inclusive. The histogram should be output to the system console and formatted as shown below.

Hotel Name	Rating
Hotel 1	**********
Hotel 2	******
Hotel 3	********
Hotel 4	**
Hotel 5	****

3. Write a program that uses nested loops to output one or more of these patterns (or create some of your own):

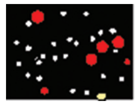
```
a)  **********      b)  *              c)       *
    **********          ***                    * *
    **********          *****                 * * *
    **********          *******              * * * *
    **********          *********           * * * * *
```

4. Write a program that asks the program users for the country in which they were born and their salary for the each of the last 12 months. Output each month's salary, as well as the total pay for that year in the format of their local currency, to the system console, properly annotated.

5. Write a program that outputs 25 random integers to the system console that are within a range (minimum value and maximum value) specified by the user.

6. Write a program to simulate the toss of two dice. Every time the user clicks the OK button on a message generate two random outputs between 1–6, as well as the sum of the two dice. If the total is 7 or 11, output *You win*, otherwise output *Better luck next time*.

7. Write a graphical application that displays 650 of the 2,500 stars that can be seen in the night time sky. The stars will be drawn on the game board as filled ovals whose diameter is a random number between one and three pixels. There will be 400 white, 200 yellow, and 50 red stars, positioned at random (x, y) locations on a black-colored game board. You can change the color of the game board by invoking the `Component` class's `setBackground` method in the `main` method and passing it the color black.



8. Write the application described in Programming Exercise 7 using three nested loops to draw the stars.

9. Write a graphical application to simulate a journey to the sun by Captain Burk. Before the game board is displayed, the captain will be required to enter the noncase sensitive password "SS" (short for Starship). Then, he will be asked to enter the tonnage of each item in his cargo. When a -1 is entered, output the total weight of the cargo to a message box and display the game board described in Programming Exercise 7 or 8 with a 50-pixel-diameter sun positioned at the center of the game board. The sun will be a yellow instance of a `HeavenlyBodies` class you will add to the application that contains:

   - The four data members of a heavenly body: its (x, y) location coordinates, its diameter, and its color
   - A four-parameter constructor
   - A `show` method, and `set` and `get` methods for all the data members

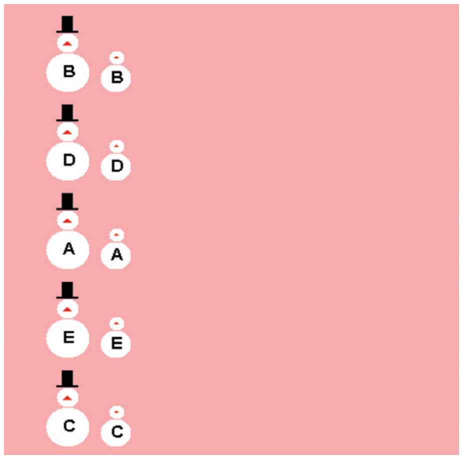
   Use an input dialog box for all input.

10. Write the graphical application described in Programming Exercise 9 expanded to include these features:
   - Before the game board is displayed, the captain will be asked how many (of a maximum of three) planets to add to the night sky and then asked to enter the location and diameter of each planet. The color of the three planets will be red, green, and blue, respectively, and they will be instances of `HeavenlyBodies` displayed on the game board.
   - When the Start button is clicked, the diameter of the sun should increase by 2 pixels every 20 milliseconds to simulate the Captain Burk's journey to the sun.
   - When the Start button is clicked a white comet (a `HeavenlyBodies` object) will travel from the upper-left to the lower-right corner of the gameboard with its diameter increasing from 3 to 50 pixels.

11. Using the skills developed in this chapter, continue the implementation of the parts of your game requiring knowledge of loops. Be sure to add this feature:
   - Do not permit the game to be played until the case-sensitive password "gp" (game player) is entered. After three unsuccessful password entries, output the statement *passwords are case sensitive* and terminate the program by invoking the `System` class's `exit` method.

Chapter 6

Arrays



In this chapter

In this chapter, we will introduce the concept of an array and the powerful features of the construct that make it a part of most programs. These features include the ability to store and retrieve large data sets, and, when combined with the concept of a loop, these data sets can be processed with only a few instructions. Array processing algorithms such as sorting, searching, and copying will be discussed and implemented, as will algorithms introduced in Chapter 4 for inserting and deleting items stored in a disk text file. We will also explore the API methods that implement many of the classical array processing algorithms.

One-dimensional arrays, which can be used to store a list of items, will be discussed as well as multi-dimensional arrays, and we will use two-dimensional arrays to organize data in tables as rows and columns.

After successfully completing this chapter you should:

- Understand the advantages and importance of using arrays
- Be familiar with the Java memory model used to store arrays
- Be able to construct and use arrays of primitives and objects
- Understand and be able to implement the algorithms used to search an array, sort it, and find the minimum and maximum values stored in it
- Be familiar with and be able to use the array-processing methods in the API
- Understand the concept of parallel arrays and use them to process data sets
- Know how to use arrays to insert, delete, or update data items stored in a disk file
- Be able to apply array techniques to game programs

6.1 THE ORIGIN OF ARRAYS

The machines we call computers received their name because the first operational versions of these machines were primarily used by mathematicians to perform rapid computations on large data sets. They were machines whose task was to compute; they were computers. However, long before computers were operational, mathematicians were using subscripted variables, such as x_2 or x_4, to represent the data used in their formulas and calculations, so it was natural for them to want to use these subscripted variables in the formulas evaluated by these early computing machines.

To facilitate the writing of these subscripted variables into a program, the designers of FORTRAN (which stands for *For*mula *Tran*slation), the first high level programming language used by mathematicians, included a construct that modeled subscripted variables. The construct was named *array*. Thus, the computer concept of an array has its roots in the mathematical model of subscripted variables.

6.2 THE CONCEPT OF ARRAYS

Consider a program that processes five people's ages stored in the integer memory cells `age0`, `age1`, `age2`, `age3`, and `age4`. The declaration of these variables would be rather straightforward:

```
int age0, age1, age2, age3, age4;
```

But suppose that instead of processing five people's ages, the program processed five million people's ages. Although the declaration syntax for the five million memory cells would still be straight forward, it would be quite lengthy and very time consuming to write. In fact, a good typist would take more than a month to type just the variable declarations for this program, assuming the typist typed continuously for eight hours each day without stopping to eat. (This, by the way, is a violation of the federal labor laws.) Using the construct array, the same typist could type the declaration of the five million memory cells in seconds.

That's all an **array** is: a technique used to declare memory cells, which is rooted in the mathematical concept of subscripted variables.

Definition

An **array** is a programming concept used to declare groups of related memory cells in which each member of the group has the same data type, the same first name, the array's name, and a unique last name called an **index**.

The memory cells are related in the same way that our integer memory cells `age0` through `age4` were related: each one stores a person's age, or perhaps a person's weight, or perhaps an address of a snowman game piece. In Java, The unique last names, the indexes (or indices), are always sequential integers beginning with zero (i.e., 0, 1, 2, 3, 4, …). In addition, Java syntax requires that the unique last name is enclosed in open and close brackets, for example, [2].

Figure 6.1 shows ten memory cells used to store people's ages. The five memory cells on the left were declared to be five separate integer variables with the statement

```
int age0, age1, age2, age3, age4;
```

The five memory cells on the right were declared to be part of a five-member or element array named `age`.

	Non-array	Array	
age0	0	0	age[0]
age1	0	0	age[1]
age2	0	0	age[2]
age3	0	0	age[3]
age4	0	0	age[4]

Figure 6.1
Storage allocated to five integer variables and to a five-element array named age.

As shown in Figure 6.1, the amount of storage allocated to the integer variables on the left side of the figure is the same as the amount of storage allocated to array elements shown on the right side of the figure: five distinct integer memory cells. From a memory-allocation viewpoint, the only difference in the way memory is allocated to the memory cells that make up the elements of an array is that the array elements are always allocated as contiguous memory; that is, if each memory cell occupied four bytes of storage, and `age[0]` was stored in bytes 100–103, the memory allocated to the subsequent four elements of the array would begin at byte addresses 104, 108, 112, and 116. (In contrast, the five integer variables might be stored in different locations scattered around memory.)

Array elements can be used in our programs anywhere it is syntactically correct to code the name of a memory cell: in input and output statements, in arithmetic and logic expressions, on the left side of an assignment operator, and as arguments and parameters. To use them, we simply code their complete names. For example, the statements on the left and right sides of Figure 6.2 are equivalent, although they are syntactically different because the statements on the right side of the figure use the array construct.

```java
age3=new Scanner(System.in).nextInt();
age3 = age3 + 1;
System.out.println("Your age is"  +
                    age3);
if(age3 >= 18)
{
   System.out.println("You can " +
                      "Drive now");
}

double avgAge = averageTwo(age0,
                           age1);
```

Without arrays

```java
age[3]=new Scanner(System.in).nextInt();
age[3] = age[3] + 1;
System.out.println("Your age is"  +
                    age[3]);
if(age[3] >= 18)
{
   System.out.println("You can " +
                      "Drive now");
}

double avgAge = averageTwo(age[0],
                           age[1]);
```

With arrays

Figure 6.2
Equivalent statements with and without the use of arrays.

Although the syntax involved in using arrays is a bit more cumbersome because of the coding of the open and close brackets, as previously mentioned, they do give us the ability to rapidly declare large numbers of variables. In addition, as we will see later in this chapter, when arrays are used inside of loops they also give us the ability to process large data sets with just a few lines of code. For these two reasons, most programs use arrays.

6.3 DECLARING ARRAYS

In Java, all arrays are stored inside of an object. Although we most often state that we are "declaring an array," it is more accurate to state that we are "declaring an object that contains an array." In fact, as we shall see, the object contains not only the array but an also an integer data member named `length`.

The syntax used to declare an array object is similar to the syntax used to declare non-array objects in that a reference variable is declared that will refer to the array object, and the keyword **`new`** is used to construct the object. Where they differ is that the array-object declaration syntax also includes a set of brackets to indicate that the reference variable will refer to an array object, and the number of elements the array will contain (called the size of the array) is enclosed in another set of brackets. The generalized syntax is:

```
aType[] arrayName = new aType[arraySize];
```

where:

`aType` is the type of the elements of the array

`arrayName` is the name of a reference variable that will store the address of the array object, also considered to be the name of the array

`arraySize` is the number of elements in the array

To declare an array object that could store five integer ages we would write:

```
int[] ages = new int[5];
```

This statement allocates the memory shown in Figure 6.3. Not only is the storage for the array's elements allocated inside the object, but an integer named `length` is allocated and initialized to the size of the array. The index of the first element of an array is always zero, and the indexes of the remaining elements of an array are assigned sequential integer values in ascending order. This implies that the index of the last element of an array is always one less than the size of the array. In a five-element array, `ages[5]` does not exist, which is somewhat counterintuitive, and attempting to access it results in a runtime error.

NOTE *The indices of an array containing n elements are 0 through n-1, and the size of the array is n.*

Conceptually, the array object declaration would be drawn as shown on the right side of Figure 6.1, which is the way we most often visualize an array. Figure 6.3 gives a more accurate depiction of the storage allocated to the array object created by the declaration given in the figure's caption and the reference variable, `ages`, that refers to the array object.

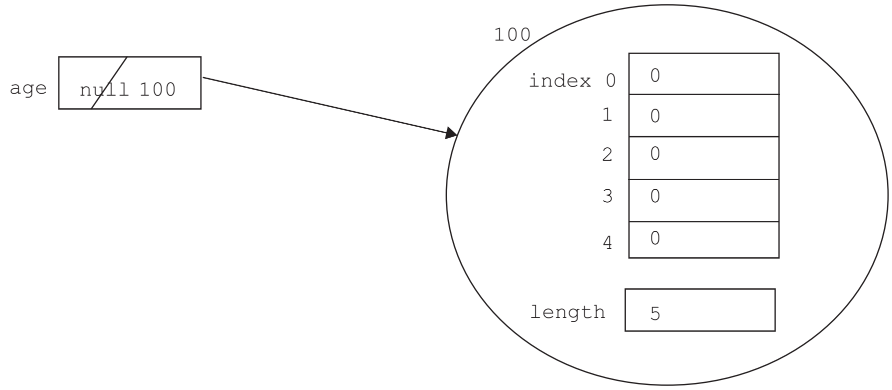


Figure 6.3
The array object created by the statement `int[] ages = new int[5];`.

When an array object is created, the elements of the array are initialized to their default values (e.g., zero for an array of integers), and the array object is assigned an address (address 100 in Figure 6.3). The data member `length` is initialized to the size of the array. For example in Figure 6.3, `length` stores the value `5` inside the array object `ages`.

The data member `length` is a public data member, so rather than using a `get` method to access it, it can be accessed by coding the name of the array object followed by `length`, preceded by a dot. The following code fragment outputs a *5*, the size of the array `ages`:

```
int[] ages = new int[5];
System.out.println(ages.length);
```

The data member `length` is a final variable and cannot be assigned a value. The following code fragment results in a translation error:

```
int[] ages = new int[5];
ages.length = 23; //syntax error: can't re-assign a final constant
```

6.3.1 Dynamic Allocation of Arrays

An array object, like any other object, can be allocated dynamically during the execution of the program. As we have learned, to do this we most often use the two-line object declaration syntax. The first line is used to declare the reference variable that will refer to the object, and good programming practice dictates that this line is coded at the top of the method or class in which the array will be used. The second line of the syntax is used to allocate the object and set the reference variable pointing to it. This line is normally coded further down in a method.

The splitting of the array object declaration syntax permits the size of the array to be determined by the processing the program performs. For example, the size of the array could be read from the first line of a disk file that also contains the data that will be stored in the array, or the size of the array could be input by the user. For example:

```
int[] ages;
String sSize = JOPtionPane.showInputDialiog("How many ages" +
                                            "will be entered?");
```

```
int size = Integer.parseInt(sSize);
ages = new int[size];
```

Many applications, in which the number of data items to be processed is determined at run time, would be very difficult to code without the use of arrays. For example, consider an application that outputs a set of input data in reverse order. This requires declaring a variable for every data item because they must all be saved until the last data item is input and then output. Because the number of inputs is not known until runtime, without the use of arrays, we would have to guess the maximum number of inputs, allocate that number of variables, and keep our fingers crossed that we did not guess too low.

The fact that the `length` data member of an array object cannot be changed is consistent with the fact that, in Java, the size of an array cannot be changed. As is the case with all objects, the reference variable that refers to the object can be assigned to another object. In the case of an array, we can make use of this fact to effectively resize the array at runtime by assigning the reference variable to the address of a smaller, or a larger, array object.

For example, an array initially sized to five elements could be made to refer to a new array object whose size is based on a user input.

```
int[] data = new int[5];
               :
               :
String sSize = JOPtionPane.showInputDialiog("How many ages" +
                                                    "will be entered?");
int size = Integer.parseInt(sSize);
data = new int[size];
```

Assuming the user entered a "3" in response to the above prompt, Figure 6.4 shows the changes in the contents of the reference variable `data` and the array object that `data` refers to, resulting from the execution of the above code. It should be noted that if the original five-element array contained five people's ages, these ages would be lost after the dynamic allocation.

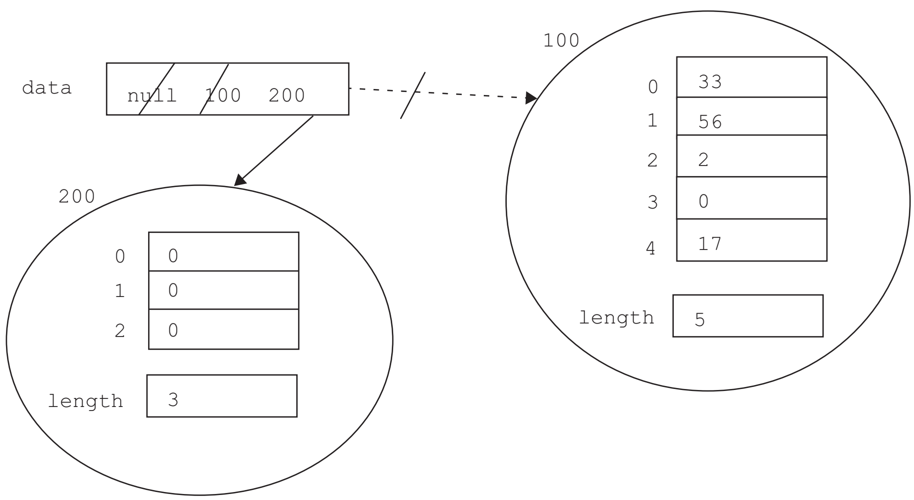


Figure 6.4
The effect of the statement data = `new int[3];`.

As shown in Figure 6.4, the five-element array object's address is overwritten with the address of the new three-element array object, causing the storage allocated to the five-element array to be reclaimed by the Java runtime memory manager. In Java, the storage allocated to objects that are not referred to by a reference variable is reclaimed for use by other programs. In Section 6.9, we will discuss techniques for transferring the values into a resized array when it is created.

6.4 ARRAYS AND LOOPS

Using arrays inside loops gives us the ability to process large data sets with just a few lines of code. This is because the index used to specify which element of the array is being processed can not only be a numeric literal (e.g., `a = age[2];`), but it can also be an integer variable. The only restriction on the integer variable is that the value stored in it must be a valid element number of the array. The following code segment outputs the third element of the array `price` twice. When the last statement executes, the current contents of the variable `index` is fetched, substituted for the variable `index`, and the output is performed.

```
double[] price = new double[100];
int index = 2;

System.out.println(price[2]);
System.out.println(price[index]);
```

This array feature is commonly used with the loop variable of a `for` statement as the array index. Using this approach, the code to decrease the price of each of the 10,000 items a department store sells by 10% in preparation for its annual Labor Day sale can be coded in just two lines of code:

```
for(int i = 0; i < 10000; i++)
{
    salePrice[i] = price[i] * 0.9;
}
```

The first time through the loop the variable `i` stores the value 0, and `salePrice[0]` is computed. The second time through the loop `i` stores the value 1, and `salePrice[1]` is computed. This process continues until finally `salePrice[9999]` is computed.

Two common mistakes are made when processing arrays inside of loops, both of which are syntactically correct:

- the loop variable is initialized to 1 instead of to 0
- the Boolean condition is incorrectly coded using the <= operator instead of <

The first mistake stems from the fact that most of us begin with 1 when we count: 1, 2, 3, etc., so our natural tendency is to initialize the loop variable to 1 instead of 0. When this mistake is made, the first element of the array (element zero) is not processed.

Coding the Boolean condition incorrectly is the most common mistake. When all the elements of the array are to be processed, our code is much more understandable if we use the size of the array, `price.length`, in the Boolean condition. However, when we do this, we must use the less than (<) operator in the condition (e.g., `i < price.length`). Unfortunately, most novice programmers, intent on processing the last element of the array, use the <= operator, and the last iteration of the loop generates an index that is one greater that of the last element of the array (e.g., 5 for a five-element array). The result is a runtime error indicating that the program generated an `ArrayIndexOutOfBoundsException`. This error occurs whenever a program uses an array index that is not in the range 0 to one less than the array's size.

Figure 6.5 presents the application `ArraysAndLoops` that uses many of the array concepts discussed thus far in this chapter to compute, and output, the sale price of a group of input items. It accepts the prices of a set of items to be placed on sale, then computes and outputs the sale price of the items. A sample set of inputs and the corresponding outputs produced by the program are given at the bottom of the figure.

```
import javax.swing.*;
import java.text.NumberFormat;

public class ArraysAndLoops
{
  public static void main(String[] args)
  {
    double[] price, salePrice;
    String s;
    int size;
    NumberFormat fm = NumberFormat.getCurrencyInstance();
    s = JOptionPane.showInputDialog("How many sale items?");
    size = Integer.parseInt(s);
    price = new double[size];
    salePrice = new double[size];

    for(int i = 0; i < size; i++)
    {
      s = JOptionPane.showInputDialog("Enter item " + (i + 1) +
                                      " 's price");
      price[i] = Double.parseDouble(s);
    }

    for(int i = 0; i < price.length; i++)
    {
      salePrice[i] = price[i] * 0.9;
      System.out.println("The sale price of item " + (i + 1)  +
                         " is " + fm.format(salePrice[i]));
    }
  }
}
```

Inputs:
5
10.00
20.00
30.00
40.00
50.00
Outputs:
The sale price of item 1 is $9.00
The sale price of item 2 is $18.00
The sale price of item 3 is $27.00
The sale price of item 4 is $36.00
The sale price of item 5 is $45.00

Figure 6.5
The application **`ArraysAndLoops`** and a set of inputs and corresponding outputs.

After the user enters the number of items to be placed on sale (line 12), two array objects are dynamically allocated on lines 14 and 15, and their addresses are assigned to the reference variables `price` and `salePrice`. These variables were declared on line 8.

The program uses two `for` loops that begin on lines 17 and 24. The first loop accepts the input of the non-sale prices, and the second loop computes and outputs the sale prices. The loop variable, `i`, of the `for` loop that begins on line 17 is used to change the element of the array `price` (line 21) that stores the parsed input. The second `for` loop, which begins on line 24, uses its loop variable to index its way through the array `price` and the array `salePrice` (line 26) as it computes the new values of the `salePrice` array.

The first loop uses the variable `size,` which was used to size both arrays on lines 14 and 15, in its Boolean condition. The second loop uses the `length` data member of the array object `price` in its Boolean condition. Either approach can be used. However, the latter approach is preferred because it more clearly indicates that the entire array is being processed within the loop, and eliminates the chance that an incorrect variable (other than size) would be coded in the Boolean condition. The second approach is also preferred when the array is passed into a method that will process the array's contents for reasons that we will discuss in Section 6.6. Both `for` statements correctly use the less than operator (<) in their Boolean conditions.

6.5 ARRAYS OF OBJECTS

Technically speaking, there is no way to declare an array of objects. The elements of an array cannot be objects; they can only be primitive or reference variables. However, when the array elements are reference variables, each element of the array can contain the address of an object. When this is the case, we often say that we have "an array of objects" because it is easier to say than "an array of reference variables that refer to objects" (which is what we actually have).

Leaving aside the technical jargon, when we set each element of an array of reference variables to point to an object, we can rapidly process all of the objects by indexing through the array of reference variables. In addition, just as it was easy to declare a large number of variables using arrays, we can easily declare a large number of objects using arrays of reference variables.

The first step in applying the power of arrays to programs that process objects is to declare (an array object that contains) an array of reference variables. The second step is to declare the objects and set their addresses into the elements of the array. The syntax used to declare the array of reference variables is the same syntax used to declare an array of primitive variables. The following declaration creates an array of reference variables that could refer to five `Snowman` objects:

```
Snowman[] sm = new Snowman[5];
```

The storage allocated by this declaration is shown in Figure 6.6. Because the array contains reference variables, they are initialized to the default value of a newly created reference variable: `null`. Otherwise, the figure is identical to Figure 6.3, which shows the storage allocated when an array of five integers is declared.

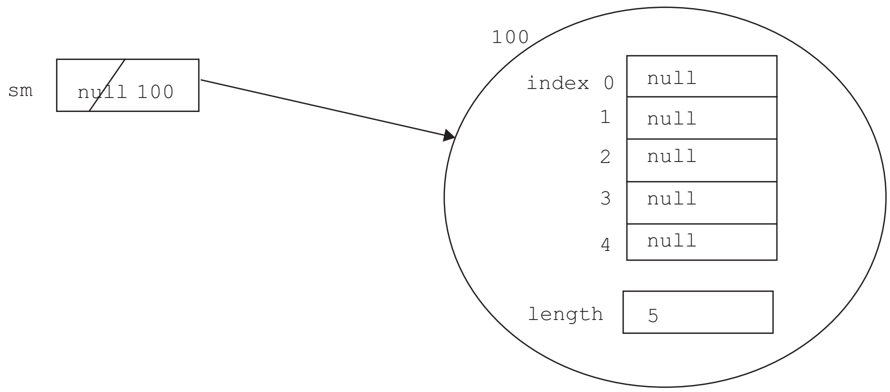


Figure 6.6
The storage created by the declaration **`Snowman[] sm = new Snowman[5];`**.

As shown in Figure 6.6, the declaration of the array object does not allocate any `Snowman` objects. To do this, we have to invoke a constructor in the `Snowman` class and set the returned address of the newly constructed `Snowman` into an element of the array. Assuming the class has a two-parameter constructor, one way to do this is to write five declaration statements:

```
sm[0] = new Snowman(50, 100);
sm[1] = new Snowman(100, 100);
sm[2] = new Snowman(150, 100);
sm[3] = new Snowman(200, 100);
sm[4] = new Snowman(250, 100);
```

Assuming the constructor's parameters are the (x, y) location of a `Snowman` object, our five newly created snowmen will be standing shoulder to shoulder (at x = 50, 100, 150, 200, and 250) when they are drawn on the game board. An equivalent but more efficient way to construct the five snowmen would be to place the invocation of the constructor inside a loop. The use of a loop is the preferred coding technique, which we would quickly realize if we had to declare a group of 5,000 snowmen.

```
for(int i = 0; i < 5; i++)
{
   sm[i] = new Snowman(50 +  i * 50, 100);
}
```

Because the loop variable is used as the index into the array `sm`, `sm[0]` receives the address of the first `Snowman` created inside the loop. During each additional pass through the loop, the next sequential element of the array receives the address of a newly created `Snowman`. In addition, the loop variable is used to change the x coordinate of the snowmen, using the expression (50 + i * 50), each time through the loop. The storage created after the loop completes its execution is shown in Figure 6.7.

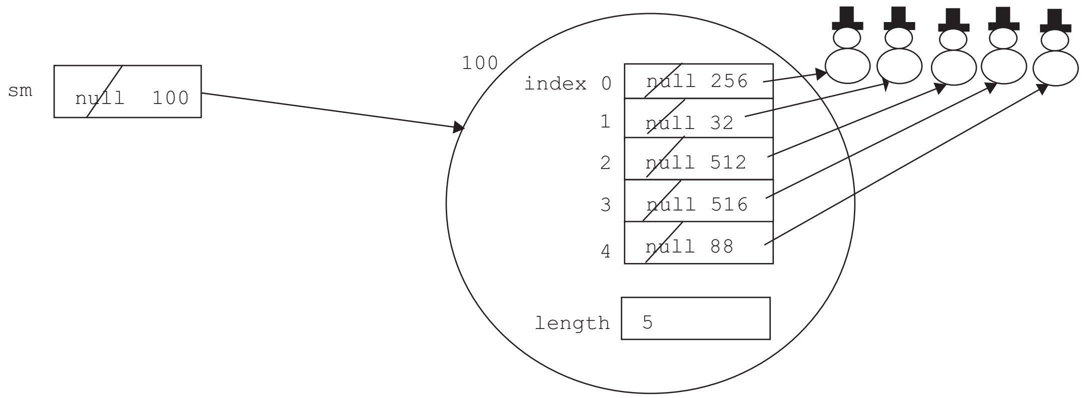


Figure 6.7
An array of five reference variables pointing to five **`Snowman`** objects.

6.5.1 Processing an Array's Objects

In Section 6.4, we learned that large primitive data sets could be processed with just a few lines of code using the concepts of arrays and loops. To accomplish this, the data set was stored in an array of primitive variables, and the processing instructions were coded inside a loop. The loop variable was used as the index into the array, which caused the processing instruction(s) to operate on a different element of the array during each pass through the loop.

Similarly, we can process large sets of objects with just a few lines of code by storing the objects' addresses inside an array of reference variables and then perform the processing on each object inside a loop. The only difference is that instead of performing the processing on the array elements themselves, the array elements are used to invoke the class's processing methods on the objects they reference.

For example, the code to add one to each of five people's ages stored in an array of integers named `ages` is very similar to the code that moves each of five snowmen stored in an array of objects named `sm` one pixel to the right. The following code fragment illustrates the similarities:

```
int x;
for(int i = 0; i < 5; i++)
{
  ages[i] = ages[i] + 1; //increment the ages

  x = sm[i].getX(); //move the snowmen
  sm[i].setX(x + 1);
}
```

Each time through this loop, the loop variable is used to change the element of the two arrays involved in loop's processing instructions. In the case of the integer array, the value stored in one of the elements of the array, `ages`, is incremented by one; that is, the contents of the array `ages` is changed. However, the loop processing does not change the contents of the array `sm`. Rather, it uses the contents of the array `sm` to specify which `Snowman` object will be changed (operated on) by the `getX` and `setX` methods during each pass through the loop. In this case, the `x` data member of each `Snowman` object is increased by one.

That is not to say that the contents of an array of reference variables cannot be changed inside a loop. As we have already seen, this is done when objects are constructed and the default `null` values stored in the elements of the array are overwritten with the location of the newly constructed (`Snowman`) objects. Conversely, all five snowmen could be eliminated from a game by overwriting their addresses stored in the array with the value `null`:

```
for(int i = 0; i < 5; i++)
{
  sm[i] = null;
}
```

This would cause the Java memory manager to recycle the storage allocated to the five `Snowmen` objects and make it available to other programs running on the system.

The game application in Figure 6.8 uses the concepts discussed in this section to conduct a parade of eight snowmen whose class is shown in Figure 6.9. The output produced by the program when it is launched and the output produced several seconds after the start button is clicked are shown on the left and right sides of Figure 6.10, respectively.

An array of reference variables named `parade` that will be used to store the addresses of eight `Snowman` objects, is created on line 7 of Figure 6.8. When the game is launched, the snowmen are displayed along a left-to-right downward-sloping diagonal (as shown in Figure 6.10a), until the Start button is clicked. Then they parade around the game board reflecting off its boundaries, eventually coming to the positions shown in Figure 6.10b.

The creation, display, animation, and reflection of the snowmen are performed inside four loops. Within each iteration of these loops, a different snowman is processed because the loop variable is used as an index into the `parade` array. The first of these loops (lines 11–14 of Figure 6.8) is used to create the snowmen and place the addresses of these eight objects in the reference variable array `parade`. On line 13, the loop variable, `i`, is used inside the argument list sent to the `SnowmanV7` class's two-parameter constructor to calculate each snowman's initial (x, y) location along a downward-sloping diagonal. During each iteration of the loop that begins on line 21, a different snowman is displayed on the game board at its current (x, y) location.

The remaining two loops, which begin on lines 31 and 41, move the snowmen around the game board and bounce (reflect) them off the vertical and horizontal boundaries of the game board. The loops are coded inside the `timer3` call back method (lines 27–56), whose interval is set to 20 milliseconds on line 15. As a result, every 20 milliseconds (1/50th of a second), the game environment invokes this method, and the loops are executed. After the `timer3` method completes its execution, the game environment invokes the application's `draw` method (line 19), which displays the snowmen at their new (x, y) position.

The `for` loop coded on lines 31–39 of the application performs the animation of the snowmen. By using the loop variable, `i`, as an index into the parade array, each snowman's x and y position is fetched (lines 33 and 36), incremented by their corresponding speed data members (lines 34 and 37), and set to their new values (lines 35 and 38). Being coded inside the `timer3` method, this code changes each snowman's (x, y) position every 20 milliseconds. The rapid repositioning and redrawing of the snowmen (every 1/50th of a second) gives the appearance of continuous motion.

The `for` loop coded on lines 41–55 of Figure 6.8 performs the reflection of the snowmen off the boundaries of the game board. The two data members, `xSpeed` and `ySpeed` were added to the class `SnowmanV7` (Figure 6.9, lines 7 and 8) along with their corresponding `set` and `get` methods (lines 41, 45, 49, and 53) to perform this reflection. The loop variable, `i`, is used inside two `if` statements (that begin on lines 43 and 49 of Figure 6.8) to index into the `parade` array. Their Boolean conditions determine when a snowman's current (x, y) position is at or beyond the vertical (line 43) and horizontal (line 49) boundaries of the game board.

When this is the case, the snowman's speed is fetched (lines 45 and 51), its sign is reversed (lines 46 and 50), and the new value is set into the snowman's speed data member (lines 47 and 53). Then, during the next execution of the `timer3` method, when each snowman's speed is used to reposition it on the game board, those that reached a game board edge appear to bounce off the edge because the sign of their speed has been reversed.

```
1    import java.awt.Graphics;
2    import edu.sjcny.gpv1.*;
3
4    public class SnowmanParade  extends DrawableAdapter
5    { static SnowmanParade ge = new SnowmanParade();
6      static GameBoard gb = new GameBoard(ge, "Snowman Parade");
7      static SnowmanV7[] parade = new SnowmanV7[8];
8
9      public static void main(String[] args)
10     {
```

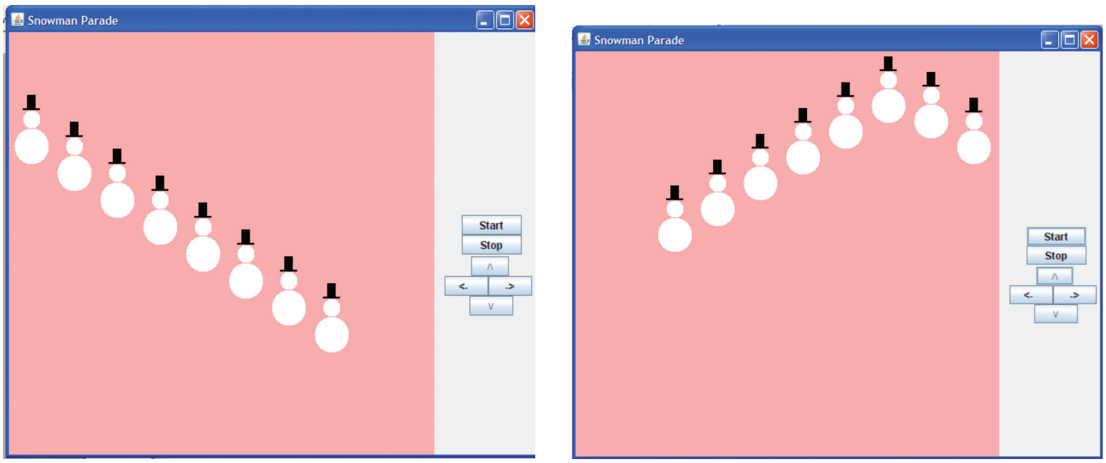
```
    for(int i=0; i < parade.length; i++) //create each snowman
    {
      parade[i] = new SnowmanV7(10 + i * 50 , 100 + i * 30);
    }
    gb.setTimerInterval(3, 20);
    showGameBoard(gb);
  }

  public void draw(Graphics g) //draw each snowman
  {
    for(int i = 0; i < parade.length; i++)
    {
      parade[i].show(g);
    }
  }

  public void timer3()
  {
    int x, speed, y;

    for(int i = 0; i  <parade.length; i++) //move each snowman
    {
      x = parade[i].getX();
      x = x + parade[i].getXSpeed();
      parade[i].setX(x);
      y = parade[i].getY();
      y = y + parade[i].getYSpeed();
      parade[i].setY(y);
    }

    for(int i = 0; i < parade.length; i++) //reflect each snowman
    {
      if(parade[i].getX() >= 460 || parade[i].getX() <= 6)//x
      {
        speed = parade[i].getXSpeed();
        speed = -speed;
        parade[i].setXSpeed(speed);
      }
      if(parade[i].getY() >= 420 || parade[i].getY() <= 30)//y
      {
        speed = parade[i].getYSpeed();
        speed = -speed;
        parade[i].setYSpeed(speed);
      }
    }
  }
}
```

Figure 6.8
The application **SnowmanParade.**

```
import java.awt.*;

public class SnowmanV7
{
  int x;
  int y;
  int xSpeed = 2;
  int ySpeed = 2;

  public SnowmanV7(int x, int y)
  { this.x = x;
    this.y = y;
  }

  public void show(Graphics g) // g is the game board object
  { g.setColor(Color.BLACK);
    g.fillRect(x + 15, y, 10, 15); // hat
    g.fillRect(x + 10, y + 15, 20, 2); // brim
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y + 17, 20, 20); // head
    g.fillOval(x, y + 37, 40, 40); // body
    g.setColor(Color.RED);
  }

  public int getX()
  { return x;
  }

  public void setX(int newX)
  { x = newX;
  }

  public int getY()
  { return y;
  }

  public void setY(int newY)
  { y = newY;
  }

  public int getXSpeed()
  { return xSpeed;
  }

  public void setXSpeed(int newXSpeed)
```

```
    { xSpeed = newXSpeed;
    }

    public int getYSpeed()
    { return ySpeed;
    }

    public void setYSpeed(int newYSpeed)
    { ySpeed = newYSpeed;
    }
}
```

Figure 6.9
The class **SnowmanV7**.



(a) Initial output (b) Output several seconds after Start is clicked

Figure 6.10
The output of the application **SnowmanParade**.

6.6 PASSING ARRAYS BETWEEN METHODS

As discussed in Section 3.8, reference variables can be part of a worker method's parameter list. This gives us the ability to pass the location of objects declared in a method into a worker method it invokes. Knowing the object's location enables the worker method to perform some processing on the object by referring to it using the parameter name that received the object's location. In addition, when the returned type of a non-void worker method is the name of a class, the worker method can return the location of *one* object in that class to the method that invoked it.

Because arrays are stored in objects, the ability to pass reference variables to and from worker methods also gives us the ability to pass arrays to and from worker methods. To do this, we simply pass the array object's reference variable to and from the worker method. While there are no conceptual differences to consider when passing array and non-array objects to and from worker methods, there are some minor syntactical differences in the signature of the worker method.

In the remainder of this section, we will discuss these differences and present examples of passing arrays to and from worker methods. We will begin with a discussion of passing arrays to a worker method and conclude with a discussion of returning an array from a worker method.

6.6.1 Passing Arrays of Primitives to a Worker Method

Consider the array `age`, shown on the left side of Figure 6.11, which stores the ages of five people that have the same birthday. The following code fragment defines and initializes this array and passes it to the static worker method `birthday` coded in the class `Party`.

```
int[] age = new int[5];
for(int i=0; i < age.length; i++)
{
  age [i] = 21 + i;
}
Party.birthday(age); //method invocation statement
```

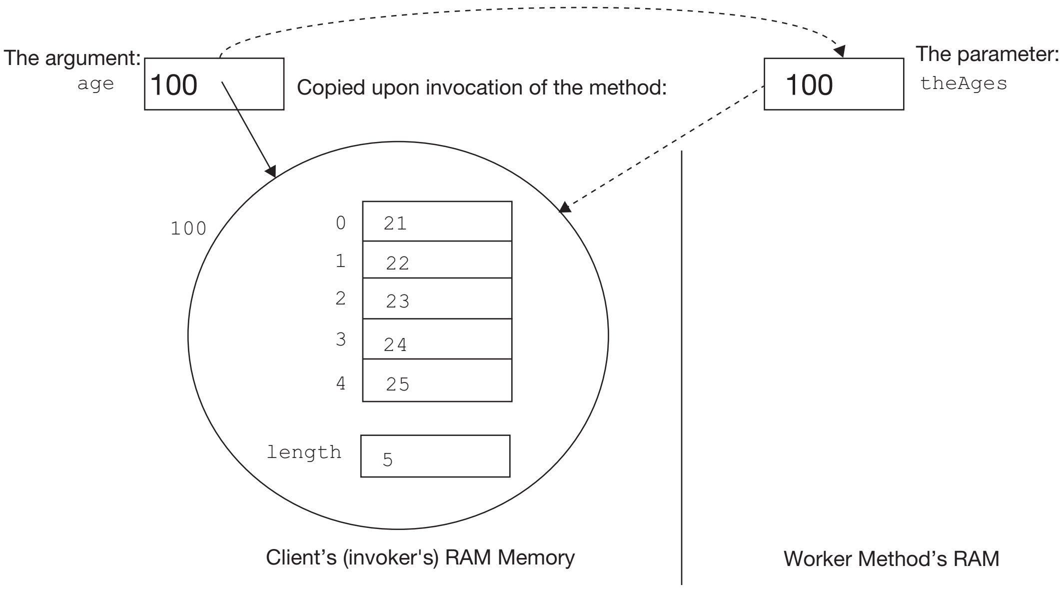


Figure 6.11
Passing an array of primitives to a method.

Looking at the method invocation statement, there is no way to tell if the argument `age` sent to the method `birthday` is a primitive variable or an array. This is because the syntax of an invocation statement used to pass an array reference to a worker method is the same syntax used to pass a primitive variable to a worker method, which is also the same syntax used to pass any reference variable to a worker method. From an invocation statement's viewpoint, there is nothing new to learn about passing arrays to worker methods.

As previously discussed, it is the syntax of the signature of the worker method that is different. When a parameter listed in a method's signature is an array, a pair of braces `[]` is coded in between the parameter's name and its type. The following is the code of the static worker method `birthday` that is passed an integer array. The work of the method is to add one to each element of the array.

```
static void birthday(int[] theAges)
{
  for(int i=0; i < theAges.length; i++)
  {
    theAges[i] = theAges[i] + 1;
  }
}
```

NOTE *When a parameter in a worker method's signature is an array, code a pair of braces* `[]` *in between the parameter's name and its type. For example:*

```
static void birthday(int[] theAges)
```

As indicated by the dashed arrows at the top of Figure 6.11, when the method is invoked and passed the argument `ages`, Java's use of value parameters copies the value stored in the argument (100) into the method's parameter `theAges`. Then the method's code is able to access the elements of the array object because its parameter, `theAges`, now stores the array's address. Effectively, while the method is executing, the array object is *shared* between the client code and the worker method it invokes. Although we normally say we are "passing an array to a method," we really should say we are "passing the *address* of an array object to a method."

NOTE *When passing an array to a worker method, the address of the array is passed to the method, and the array object is shared between the client and worker methods.*

Because the array's address is shared, if the worker method changes the contents of the array, the client code no longer has access to the original contents of the array. This is not a contradiction of the idea that value parameters prevent worker methods from changing the client's information passed to it because the information passed to the worker method is the array's *address*, not the array's *elements*. This is a subtle but important point to understand. Referring to Figure 6.11, while it is true that the worker method can change the contents of the elements of the array because `theAges` stores the object's address, it cannot change the address of the array stored in the variable `age` (which was the information passed to it).

If the worker method contained the statement below, it would lose access to the client's array object, but the client code would not. The address of the client's array object would still be stored in the variable `age`.

```
theAges = new int[20];
```

6.6.2 Passing Arrays of Objects to a Worker Method

As mentioned in Section 6.5, technically speaking, Java does not support arrays of objects. However, an array's elements can be reference variables that each store the address of an object. When this is the case, we often say that the array is "an array of objects" because it is simpler than saying the array is "an array of reference variables that point to objects." When passing an array of objects to a method, the invocation statement and the method's signature use the same syntax used to pass an array of primitive variables to a method. As discussed in the previous section, the only indication that arrays are being passed is the inclusion of a pair of braces `[]` in between the parameter's name and its type in the signature of the worker method.

The left side of Figure 6.12 shows the array, `parade`, containing five reference variables that store the addresses of five `SnowmanV7` objects whose class is shown in Figure 6.9. To pass this array to a worker method, we take advantage of Java's value parameter implementation and pass the address of the array to the method using the same syntax we used to pass the address of an array of primitives to a method.

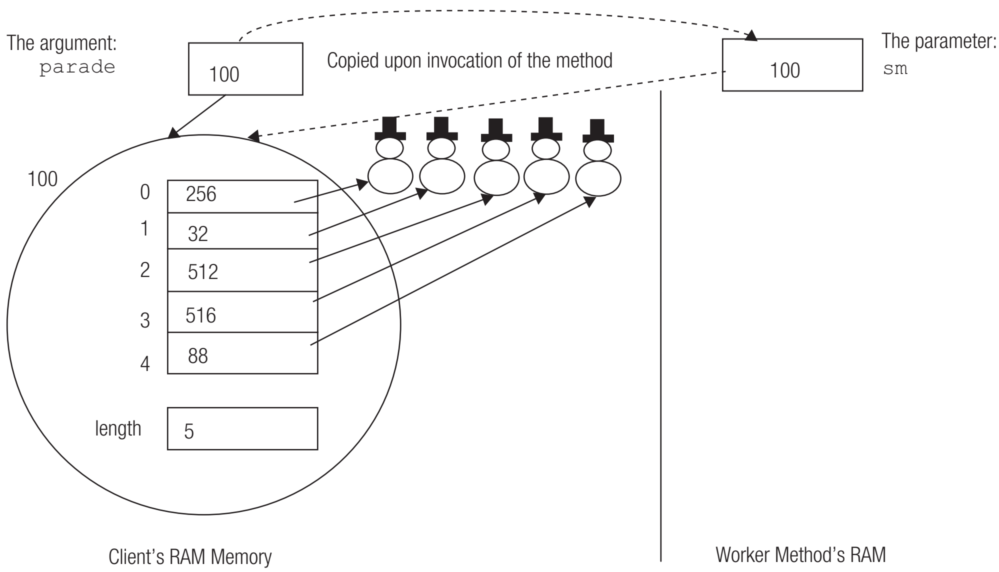


Figure 6.12
Passing the location of an array of objects to a method.

For example, suppose we wanted to use the static worker methods `move` and `bounce` coded in the class `PassingArrays` to reposition the `SnowmanV7` objects on the game board and reflect them off the edges of the board. Assuming these methods each had one parameter used to pass the array of snowmen named `parade` to the methods, the client code statements to invoke the worker methods would be:

```
PassingArrays.move(parade);
PassingArrays.bounce(parade);
```

If the parameter name in the worker methods was `sm`, the signature of the methods would be:

```
static void move(SnowmanV7[] sm);
static void bounce(SnowmanV7[] sm);
```

The dashed arrows at the top of Figure 6.12 illustrate the passing of the array's location into the method's parameter, `sm`, which permits the methods to reference the array of snowmen during their execution. The result is that while the methods are executing, the client code and the worker methods share the array of reference variables *and* the objects they refer to. If the worker methods' code writes new values into the data members of the snowmen objects, then these new values would be available to the client code after the worker methods completed their execution.

The application `PassingArrays`, shown in Figure 6.13, illustrates the sharing of the information contained in integer and object arrays with invoked methods and the methods' ability to change the contents of the array's elements and the objects they reference. Figure 6.14 shows the console output produced by the program, and Figure 6.15 shows the graphical output it produces (which is the same as that produced by the application `SnowmanParade`). The class `SnowmanV7` referred to in Figure 6.13 is the same class (shown in Figure 6.9) used in the `SnowmanParade` application.

To verify the concept that the client and worker methods share primitive arrays, we have also included the array object `ages` and the method `birthday` within the `PassingArrays` application. Referring to Figure 6.13, the application's `main` method declares an array of five integers named `ages` on line 11. These variables are initialized to the values 21 through 25 inside the `for` loop that begins on line 13. On line 17, this array is passed to the method `birthday` (lines 45–51), and the address of the array is stored in the method's parameter `theAges` (line 45). The code of the method increases each element of the array by 1 inside its `for` loop (line 49). When the method ends, the `for` loop in the method `main` (lines 18–20) outputs the contents of the array, producing the console output shown in Figure 6.14. The fact that all of the ages output by the method `main` have been increased by 1 verifies that the methods `main` and `birthday` shared the same array of integers.

The code of the graphical portion of the applications `PassingArrays` is the same as the code of the application `SnowmanParade` (Figure 6.8) except that the code that moves and reflects the snowmen (coded on lines 29–55 of Figure 6.8) have been placed inside the static methods `move` and `bounce` (lines 53–87 of Figure 6.13). In addition, invocations of these methods have been placed inside the game environment's `timer3` method, lines 41 and 42. Thus, the graphical output of the two programs is the same. When the game is launched, the snowmen are displayed along the left-to-right downward-sloping diagonal, as shown Figure 6.15a. When the Start button is clicked, they parade around the game board bouncing off its edges. Figure 6.15b shows the program's graphical output several seconds after the Start button is clicked.

The `timer3` method now consists of two statements: the invocations of the `move` and `reflect` methods (lines 41 and 42 of Figure 6.13). The address of the array `parade` (declared on line 7) is passed into the parameter `sm` of these methods, whose signatures are given on lines 53 and 68. Inside their `for` loops, the methods change the (x, y) location (lines 61 and 64) and speed data members (lines 78 and 84) of the snowmen referenced by `sm`'s elements. Because the array is shared between these methods and the `timer3` method, the new locations and speed of the snowmen have been placed into the `Snowman` array reference by `parade`. This fact is verified during the next invocation of the game environments `draw` method when the snowmen are drawn at their new locations and reflected off the edges of the game board.

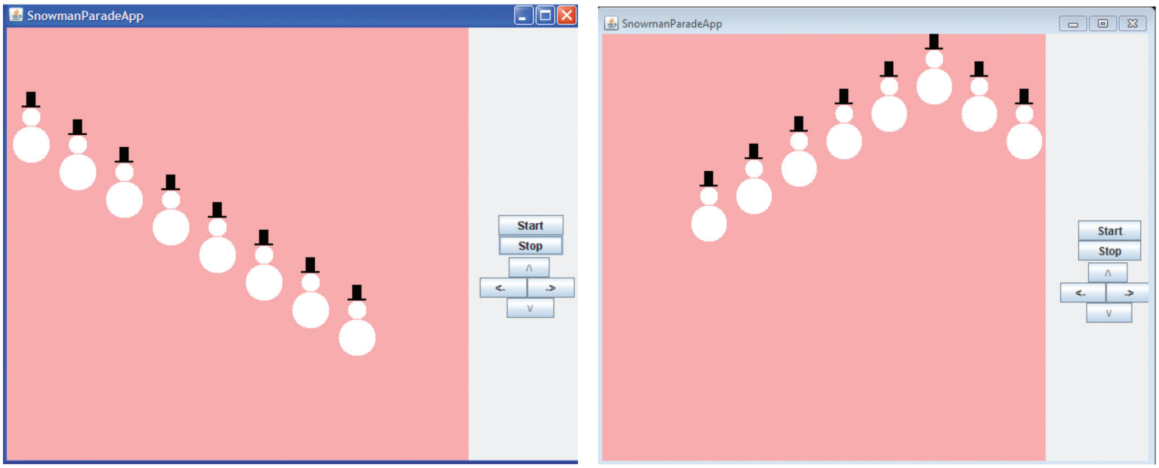
```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class PassingArrays  extends DrawableAdapter
{ static PassingArrays ge = new PassingArrays();
  static GameBoard gb = new GameBoard(ge, "Snowman Parade");
  static SnowmanV7[] parade = new SnowmanV7[8];

  public static void main(String[] args)
  {
    int[] ages = new int[5];

    for(int i = 0; i < 5; i++)
    {
      ages[i] = 21 + i;
    }
    birthday(ages);
    for(int i = 0; i < 5; i++)
    { System.out.print(ages[i] + " ");
    }

    for(int i = 0; i < 8; i++)
    {
      parade[i] = new SnowmanV7(10 + i * 50, 100 + i * 30);
    }

    gb.setTimerInterval(3, 20);
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 0; i < 8; i++)
    {
      parade[i].show(g); // show the parade at its current location
    }
  }

  public void timer3()
  {
    move(parade);
    bounce(parade);
  }

  static  void birthday(int[] theAges)
```

```
  {
    for(int i = 0; i < theAges.length; i++)
    {
      theAges[i] = theAges[i] + 1;
    }
  }

  static void move(SnowmanV7[] sm)
  {
    int x, y;

    for(int i = 0; i < 8;  i++)
    {
      x = sm[i].getX();
      x = x + sm[i].getXSpeed();
      sm[i].setX(x);
      y = sm[i].getY();
      y = y + sm[i].getYSpeed();
      sm[i].setY(y);
    }
  }

  static void bounce(SnowmanV7[] sm)
  {
    int speed;

    for(int i = 0; i < 8; i++)
    {
      if(sm[i].getX() >= 460 || sm[i].getX() <= 6)
      {
        speed = sm[i].getXSpeed();
        speed = -speed;
        sm[i].setXSpeed(speed);
      }
      if(sm[i].getY() >= 420 || sm[i].getY() <= 30)
      {
        speed = sm[i].getYSpeed();
        speed = -speed;
        sm[i].setYSpeed(speed);
      }
    }
  }
}
```

Figure 6.13
The application **PassingArrays**.

```
22 23 24 25 26
```

Figure 6.14
The console output produced by the application **PassingArrays.**



(a) Initial output (b) Output after several reflections

Figure 6.15
The graphical output of the application **`PassingArrays`**.

6.6.3 Returning an Array from a Worker Method

As discussed in Section 3.8, an object's address can be returned from a method via a `return` statement. Because arrays are stored in objects, this also gives us the ability to return the address of an array from a worker method. The only syntactical difference to consider when returning the address of an array object is in the signature of the worker method. When a method returns an array, a pair of braces `[]` is coded in between the method's name and its returned type. As is the case when any value or address is returned from a method, if the returned address is to be used by the client code that invoked the method, the client code must assign the returned address to a variable.

The following is the code of the static worker method `birthdayV2` that is passed an integer array and returns a copy of the array with all of its elements increased by one. The contents of the array `theAges` passed into the method are unchanged.

```
static int[] birthdayV2(int[] theAges)
{
  int[] newAges = new int[length.theAges]; //declares the returned array
  for(int i = 0; i < theAges.length; i++)
  {
     newAges[i] = theAges[i] + 1;
  }
     return newAges; //returns the address of the new array
}
```

The syntax of the signature of a method that returns an array of objects is the same as the syntax used in the above method's signature, except the primitive type is replaced with the name of the object's class (e.g., `SnowmanV7[]` would replace `int[]`). In Chapter 7, we discuss and present a very important example of returning an array of objects from a method.

NOTE

When a method returns an array, a pair of braces `[]` *is coded in between the method's name and its returned type in the method's signature. For example:*

```
static int[] birthday(int[] theAges)
```

The return statement only contains the name of the array without the braces. For example:

```
return newAges;
```

6.7 PARALLEL ARRAYS

Suppose you were writing a program to maintain a database of student information for a school that had 1,000 students. Specifically, three pieces of information would be stored for each student: the student's identification number, age, and grade point average (GPA). If you were an object oriented programmer, you would begin by creating a class, probably named `Student`, which contained three data members, one for each piece of information. In addition, the class would contain a constructor to construct student objects and an input method to input information into a student object. Your application could then create the database as shown below:

```
Student[] studentInfo = new Student[1000]; //1,000 Student object array

for(int i = 0; i < 1000, i++)
{
    studentInfo[i] = new Student(); //create a new Student object
    student[i].input; //input a student's information
}
```

Now suppose that your program was going to be maintained by Anna, a programmer who was not trained in object oriented programming. She is not familiar with the concept of classes, the construction of objects, data members, and the idea that a class's non-static methods (e.g., the `input` method) can operate on an object's data members.

Anna's programming training was in the alternate programming paradigm, the *procedural* paradigm, in which objects do not play a central role in the language's constructs. She is not "object friendly". Because both the procedural and object oriented paradigms include the concept of arrays, you have decided to eliminate the use of the `Student` class from your program and simply use three arrays. One array will store all the student identification numbers, another will store all the student ages, and the third array will store all the student grade point averages (GPAs). Your application would then create the database as shown below:

```
int[] id = new int[1000];
int[] age = new int[1000];
double[] gpa = new double[1000];
String sInput;
```

```
5
6  for(int i = 0; i < 1000, i++)
7  {
8    sInput = JOptionPane.showInputDialog("Enter a student's ID number);
9    id[i] = Integer.parseInt(sInput);
10   sInput = JOptionPane.showInputDialog("Enter THAT student's age);
11   age[i] = Integer.parseInt(sInput);
12   sInput = JOptionPane.showInputDialog("Enter THAT student's GPA);
13   gpa[i] = Double.parseDouble(sInput);
14 }
```

The code that is used to input the student information is an example of the use of parallel arrays. This is easily recognized when we examine the input prompts on lines 8, 10, and 12. On line 8, the user is instructed to *input a student's ID number*, implying that any of the 1,000 student IDs could be entered. Perhaps ID number 15647. However, on lines 10 and 12, the user is instructed to enter *THAT student's age* and *THAT student's GPA*. This implies that the next two entries must be the age and GPA of the student whose ID number was just entered.

Because the loop variable is used as the index into the three arrays, a student's information is stored in the *same* element of all three arrays. This is the concept of parallel arrays. Each piece of data that is associated with a particular student is stored in the same element of each of the three arrays that comprise the data set. We would not be using the concept of parallel arrays if a particular student's ID number was stored in element 3 of the `id` array, and that student's age and GPA were stored in element 24 of the `age` array and element 6 of the `GPA` array, respectively.

The name *parallel arrays* comes from the idea that if we were to draw the three arrays side by side and then draw parallel horizontal lines below and above each element of the array, as depicted in Figure 6.16, all of a student's information would be contained between two of the lines. For example, the age of the student whose ID is 76892 would be 19, and the student's GPA would be 4.0. All of Al's information, including his GPA of 1.7, would have been entered during the first iteration of the input loop. (Please study more, Al.)

	id		age		gpa	
0	15647		18		1.7	Al's info
1	3452		21		2.55	Flo's info.
2	76892		19		4.0	Bob's info.
3	34376		22		3.85	Jo's info.
4	77834		19		3.3	Ed's info.
:	:		:		:	:
999	45823		20		2.3	Jen's info.

Figure 6.16
Three parallel arrays.

It is important to remember that parallel arrays are a concept or a model, not a programming language construct. The concept is used when the programmer stores all the data for a particular entity in the same elements of two or more arrays.

Parallel arrays are used less frequently in programs coded in object oriented programming languages like Java because all of the data for a particular entity can be stored inside an object as its data members, rather than in the elements of several arrays. However, if we wanted to group several different objects together, e.g., a snowman and its child, then a set of parallel arrays of objects is a perfect way to do this.

For example, suppose that five snowmen had one snow child each. Then, if the arrays `parent` and `child` were used to store the addresses of the snowmen and the snow children, respectively, by considering the two arrays to be parallel, we could quickly locate a child's parent or locate a parent's child. The address of a parent and the address of its child would be stored in the same two elements of the arrays: `parent[2]`'s child's address would be stored in `child[2]`.

Figure 6.17 shows the graphical application `ParallelArrays` that uses three parallel arrays to associate a parent snowman with its snow child and their family name. The parent snowman's class, `ParentSnowman`, is shown in Figure 6.18, and the snow child's class, `SnowChild`, is shown in Figure 6.19.

When the program is launched, the graphical output shown in Figure 6.20a is produced. The parent snowmen are lined up vertically on the left of the game board, and the snow children are located at random locations to their right. Every child and parent has their family name (last name) displayed on their bellies. When a key is struck, the children are repositioned next to their parents as shown Figure 6.20b. Parallel arrays are used to make the association between a parent, its child, and the family name.

Three array objects are created on lines 14–16, with the first of these (the array `names`) initialized to the names of the five families (B, D, A, E, and C). Then the other two array objects are filled with references to `ParentSnowman` and `SnowChild` objects inside the `for` loop that begins on line 20. The constructors used to create the objects on lines 24 and 25 accept three arguments. The first two are the x and y location of the object, which for the children are random numbers generated on lines 22 and 23. The third parameter is the family name that will appear on the object's belly. The constructors store this name in the parent and child classes' data member `name` on line 17 of Figures 6.18 and 6.19, respectively.

The loop that begins on line 20 of Figure 6.17 establishes the arrays as parallel arrays. With each pass through the loop, the loop variable `i` is used on lines 24 and 25 as an index into the array `names` to select the family name of a parent (line 24) and its child (line 25). This name is passed to the `ParentSnowman` and `SnowChild` constructors, and the returned addresses are stored in the element `i` of the `parent` and `child` arrays. Because the same index number is used in all three arrays, a parent, its child, and their family name are all stored in the same element number of their respective arrays.

The parallel construction of the arrays makes it easy to reposition the children next to their parents, which is done in the `for` loop that begins on line 42 of the `keyStruck` call back method. Lines 44 and 45 fetch the (x, y) location of the ith child's parent, and these values are used on lines 46 and 47 to reposition the ith child to the right of its parent. Fifty pixels are added to the parent's x coordinate to move the child to the right of its parent, and 35 pixels are added to the y coordinate of the parent to account for the difference in height of the parent and child objects.

Parallel arrays are also used in the API `Graphics` class's method `fillPolygon`. This method is used on lines 31 and 28 of Figures 6.18 and 6.19, respectively, to draw the triangular noses of the parent snowmen and their children. The method has three parameters, two of which are arrays of integers:

```
public void fillPolygon(int[] xPoints, int[] yPoints , int nPoints)
```

The parameters are used to specify the x coordinates of the vertices of a polygon (`xPoints`), the y coordinates of the vertices of a polygon (`yPoints`), and the number of vertices (`nPoints`). Within the method, the two arrays are used as parallel arrays. The coordinate of the ith vertex of the polygon is assumed to be (`xPoints[i], yPoints[i]`). That is, the x and y coordinates of a vertex are assumed to be at the same element number in the `xPoints` and `yPoints` arrays. Knowing this, the two arrays `xPoly` and `yPoly` passed to the method on line 31 of Figure 6.18 have been set up as parallel arrays on lines 20 and 21. Because the desired coordinates of the three vertices of a parent's nose are (x + 20, y + 25), (x + 15, y + 30), and (x + 25, y + 30), these x and y parings are coded in the same elements of both arrays. A similar set of parings defines a child's nose on lines 21 and 22 of Figure 6.19, which are used to draw the child's nose on line 28 of that figure.

```
import edu.sjcny.gpv1.*;
import java.awt.*;
import java.util.Random;

public class ParallelArrays extends DrawableAdapter
{
  static ParallelArrays ge = new ParallelArrays();
  static GameBoard gb = new GameBoard(ge, "Parallel Object ArraysApp");
  static ParentSnowman[] parent;
  static SnowChild[] child;

  public static void main(String[] args)
  {
    String[] names = { "B", "D", "A", "E", "C"};
    parent = new ParentSnowman[5];
    child = new SnowChild[5];
    Random rn = new Random(500);
    int x, y;

    for(int i = 0;  i < 5;  i++)
    {
      x = 100 + rn.nextInt(500 - 100 - 30);
      y = 30 + rn.nextInt(500 - 30 - 30);
      parent[i] = new ParentSnowman(50, 50 + 90*i, names[i]);
      child[i] = new SnowChild(x, y, names[i]);
    }
```

```
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 0; i<5; i++)
    {
      parent[i].show(g);
      child[i].show(g);
    }
  }

  public void keyStruck(char key)
  {
    int x, y;
    for(int i = 0; i< 5; i++)
    {
      x = parent[i].getX();
      y = parent[i].getY();
      child[i].setX(x + 50);
      child[i].setY(y + 35);
    }
  }
}
```

Figure 6.17
The application **ParallelArrays**.

```
import java.awt.*;

public class ParentSnowman
{
  private int x = 8;
  private int y = 30;
  private boolean visible = true;
  private String name;

  public ParentSnowman()
  {
  }

  public ParentSnowman(int intialX, int intialY, String name)
  { x = intialX;
    y = intialY;
    this.name = name;
  }

  public void show(Graphics g) //g is the game board object
  { int[] xPoly = {x + 20, x + 15, x + 25};
```

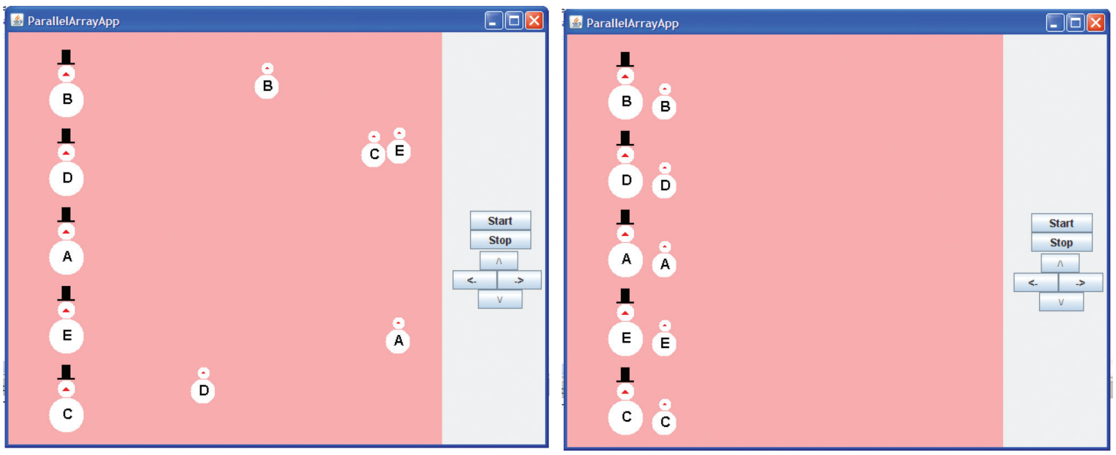
```
      int[] yPoly = {y + 25, y + 30, y + 30};

      g.setColor(Color.BLACK);
      g.fillRect(x + 15, y, 10, 15); //hat
      g.fillRect(x + 10, y + 15, 20, 2); //brim
      g.setColor(Color.WHITE);
      g.fillOval(x + 10, y + 17, 20, 20); //head
      g.fillOval(x, y + 37, 40, 40); //body
      g.setColor(Color.RED);
      g.fillPolygon(xPoly, yPoly, 3); //nose
      g.setColor(Color.BLACK);
      g.setFont(new Font("Arial", Font.BOLD, 16));
      g.drawString(name, x + 16, y + 62); //name
    }

    public int getX()
    { return x;
    }

    public void setX(int newX)
    { x = newX;
    }

    public int getY()
    { return y;
    }

    public void setY(int newY)
    { y = newY;
    }

    public boolean getVisible()
    { return visible;
    }

    public void setVisible (boolean newVisible)
    { visible  = newVisible;
    }

    public String getName()
    { return name;
    }
}
```

Figure 6.18
The class **ParentSnowman**.

```
import java.awt.*;

public class SnowChild
{
  private int x = 8;
  private int y = 30;
  private boolean visible = true;
  private String name;

  public SnowChild()
  {
  }

  public SnowChild(int intialX, int intialY, String name)
  { x = intialX;
    y = intialY;
    this.name = name;
    }

  public void show(Graphics g) //g is the game board object
  { int[] xPoly = {x + 15, x + 12, x + 18};
    int[] yPoly = {y + 5, y + 8, y + 8};

    g.setColor(Color.WHITE);
    g.fillOval(x + 8, y, 14, 14); //head
    g.fillOval(x, y + 14, 28, 28); //body
    g.setColor(Color.RED);
    g.fillPolygon(xPoly, yPoly, 3); //nose
    g.setColor(Color.BLACK);
    g.setFont(new Font("Arial", Font.BOLD, 16));
    g.drawString(name, x + 10, y + 33);  //name
  }

  public int getX()
  { return x;
  }

  public void setX(int newX)
  { x = newX;
  }

  public int getY()
  { return y;
  }

  public void setY(int newY)
  { y = newY;
  }

```

```
    public boolean getVisible()
    { return visible;
    }

    public void setVisible (boolean newVisible)
    { visible  = newVisible;
    }

    public void setName(String newName)
    { name = newName;
    }

    public String getName()
    { return name;
    }
}
```

Figure 6.19
The class **`SnowChild`**.



(a) (b)

Figure 6.20
The graphical output of the application **`ParallelArrays`**.

6.8 COMMON ARRAY ALGORITHMS

As we have seen, arrays can be used to easily declare and process large data sets. Often, the processing performed on these data sets involves searching for a particular piece of data (e.g., the snowman family whose name is C), finding the name of the snowman family that is first or last in alphabetical order, or displaying the snowman families in sorted order based on their names. Searching, finding minimums and maximums, and sorting are all array-processing algorithms that are very commonly used in programs. We will begin our study of these algorithms with the array searching algorithm.

6.8.1 Searching

As its name implies, this algorithm is used to search an array to determine the element number of the array that contains a given value. For example, it could be used to search an array containing a group of people's ages to find the element number whose value is 32. If a parallel array contained the names of the people, then the element number could be used as an index into the name array to output the name of a person who is 32 years old. In an object oriented context, it could be used to search an array of parent snowmen to find the element number of the snowman whose `name` data member contains the string "C" and then use this element number to display its family name on the game board.

An algorithm to search for, or locate, a particular value contained in an array is shown in Figure 6.21. Named the Sequential Search, it begins its search at element zero and sequentially searches through the array in element number order. The implementation on the left side of the figure searches the *integer* array `ages` for the value 32, and the code on the right searches the *object* array `parent` for an object whose `age` data member is 32. These *target* values are specified on line 1. Lines 2–12 constitute the searching algorithm. Except for the names of the arrays and the Boolean conditions on line 6, the algorithms are identical. Line 6 of the array of objects version of the algorithm (the right side of the figure) assumes that the class of the objects contains a `getAge` method to fetch an object's `age` data member.

```
1 int target = 32;
2 int elementNumber = -1;
3 boolean found = false;
4 for(int i = 0; i < ages.length; i++)
5 {
6   if(ages[i] == target)
7   {
8     found = true;
9     elementNumber = i
10    break;
11  }
12}
```

Searching an Array of Primitives

```
1 int target = 32;
2 int elementNumber = -1;
3 boolean found = false;
4 for(int i = 0; i< parent.length; i++)
5 {
6   if(parent[i].getAge() == target))
7   {
8     found = true;
9     elementNumber = i
10    break;
11  }
12}
```

Searching an Array of Objects

Figure 6.21
The Sequential Search algorithm.

The initializing value on line 1 of Figure 6.21 is the target value to be found. Line 2 initializes the Boolean variable `found` to `false`. This variable will be set to `true` if the target being searched for is found. The loop that begins on line 4 indexes its way through the array. Inside that loop, the `if` statement on line 6 determines if element `i` of the array contains the target value. If it does, `found` is set to `true` (line 8), `elementNumber` is set on line 9 to the element number that contains the target value, and the code breaks out of the loop (line 10). Subsequent code would have to examine the variable `found` before using the index stored in the variable `elementNumber` because, if the target value is not found, the value stored in the variable `elementNumber` would be out of bounds (i.e., equal to its initial value, -1).

The algorithm on the right side of Figure 6.21 can be used to locate a value stored in any primitive-type data member contained in the array's objects, as long as the class of the objects contains a `get` method to fetch the data member (which would be invoked on line 6). If the data member is not a primitive-type variable, but rather a reference variable, then the class of the object it references also must contain an `equals` method to be used in the Boolean condition on line 6. Typically, this method returns `true` when the object that invoked it is equal to the argument sent to it (the variable `target`). The `String` class contains an implementation of this method.

Assuming the data member's name was `lastName` that referenced a string, and we were searching for the name Jones, lines 1 and 6 of the algorithm would become:

```
Line 1: String target = "Jones";
Line 6: if(parent[i].getLastName().equals(target))
```

It should be noted that if several elements of the array contained the target value, the variable `elementNumber` would be set to the index of the lowest of these elements. If the highest element number of the array that contains the minimum value is desired, then the `break` statement on line 10 of Figure 2.21 would be eliminated from the algorithm. In addition, when the algorithms are implemented as a method that returns the element number of the target value, the method returns the variable `elementNumber`. The signature of the method would be:

```
public static int findValue(ArrayType[] arrayOfvalues, TargetType target)
```

where `ArrayType` and `TargetType` are the types of the array and the value being searched for, respectively. The method below is an implementation of the Sequential Search algorithm. It searches the array of `SnowChild` objects passed to its first parameter and returns the index of the first child whose name is the string passed to its second parameter. Otherwise, it returns a -1.

```
public static int findValue(SnowChild[] anArray, String target)
{
  int elementNumber = -1;

  for(int i = 0; i < anArray.length; i++)
  {
    if(anArray[i].getName().equals(target))
    {
      elementNumber = i;
      break;
    }
  }
  return elementNumber;
}
```

6.8.2 Minimums and Maximums

The algorithms to locate the minimum or maximum value contained in an array are very similar to the searching algorithm discussed in the previous section. They also use a `for` loop to search the entire array, but when the loop terminates, the variable `elementNumber` contains the element number of the minimum or maximum value in the array. When the array is an array of objects, this value is the minimum or maximum value stored in a particular data member of the objects.

The code shown on the left side of Figure 6.22 is an implementation of the algorithm to locate the *minimum* value contained in an array of primitive values (in this case, an array of people's ages). The code on the right searches the object array `parent` for the minimum value stored in one of the object's data members (in this case, the integer data member `age`). Except for lines 1, 6, and the different array names the algorithms are identical.

Lines 1 and 6 of the array of objects version of the algorithm (right side of the figure) assume that the class of the objects contains a `getAge` method to fetch the object's data member (`age`).

Both algorithms begin with the assumption that the minimum value is stored in, or is referenced by, the first element of the array. Therefore, line 1 sets the variable `min` to that value, and line 2 stores its index (zero) in the variable `elementNumber`. The `for` loop that begins on line 4 compares the value stored in the variable `min` to all of the other members of the array. If it finds a value smaller than the value stored in `min` (line 6), it saves that value in `min` (line 8) and its element number in the variable `elementNumber` (line 9). When the loop ends, `elementNumber` contains the index of the minimum value in the array.

```
1 int min = ages[0];
2 int elementNumber = 0;
3
4 for(int i = 1; i < ages.length; i++)
5 {
6   if(ages[i] < min)
7   {
8     min = ages[i];
9     elementNumber = i
10  }
11}
```

Minimum Primitive Array Value Algorithm

```
1 int min = parent[0].getAge();
2 int elementNumber = 0;
3
4 for(int i = 1; i< parent.length; i++)
5 {
6   if(parent[i].getAge() < min)
7   {
8     min = parent[i].getAge();
9     elementNumber = i
10  }
11}
```

Minimum Object Array Value Algorithm

Figure 6.22
The minimum value algorithm.

Using an approach similar to the search algorithm, the algorithm on the right side of Figure 6.22 can be used to locate the minimum value of any primitive type data member contained in the array's objects, as long as the objects' class contains a `get` method to fetch the data member (invoked on lines 1, 6, and 8). If the data member is not a primitive type variable, but rather a reference variable, then the class of the object it references also must contain a `compareTo` method to be used in the Boolean condition on line 6. Typically, this method returns a negative number when the object that invoked it is less than the argument sent to it (the variable `min`). The `String` class contains an implementation of this method.

Assuming the data member referenced a `String` object, and the data member's name was `lastName`, lines 1 and 6 of the algorithm would become:

```
Line 1: String min = parent[0].getLastName();
Line 6: if(parent[i].getLastName().compareTo(min) <  0)
```

Finally, when the array is an array of `String` objects, lines 1 and 6 of the algorithm would become:

```
Line 1: String min = parent[0];
Line 6: if(parent[i].compareTo(min) <  0)
```

It should be noted that if several elements of the array contained the minimum value, the variable `elementNumber` would be set to the index of the lowest value of these elements. If the highest index of the array that contains the minimum value is desired, then the less than operator (<) used in the Boolean expression on line 6 would be changed to the less than or equal to operator (<=). When the algorithms are coded as a method that returns the element number of the minimum value, the method returns the variable `elementNumber`. The signature of the method would be:

```
public int findMin(ArrayType[] arrayOfvalues)
```

where `ArrayType` is the types of the array elements (e.g., `double`, `String`, `SnowChild`, etc.). The following method searches the array of `SnowChild` objects passed to it and returns the index of the snow child that contains the minimum value of the primitive data member `x`.

```
public int findMin(SnowChild[] arrayOfvalues)
{
  int min = arrayOfValues[0].getX();
  int elementNumber = 0;

  for(int i = 1; i  <  arrayOfValues.length; i++)
  {
    if(arrayOfValues[i].getAge() < min)
  {
    min = arrayOfValues[i].getAge();
    elementNumber = i;
  }
  return elementNumber;
}
```

Locating Maximums

The algorithm to locate the maximum value contained in an array is the same as that used to locate the minimum value, except that the less than operator (<) used on line 6 of Figure 6.22 is replaced with the greater than operator (>), and good coding style dictates that the variable `min` be renamed `max`.

6.8.3 Sorting

There are many algorithms for sorting the elements of an `n` element array. One of the simplest is the Selection Sort algorithm. It begins by locating the minimum value contained in elements 1 to `n-1`, and if it is smaller than element zero (`j` = 0), it is swapped into element zero. Then it locates the next smallest value, and if it is smaller than element one (`j` = 1), swaps it into element one. This process is repeated for j = 2, 3, … `n-2`. When the algorithm ends, the array is sorted in ascending order.

For example, suppose we wanted to sort an array of five integers, 12, 9, 3, 4, and 11, shown in the left column of Table 6.1 in ascending order. First, we locate the smallest value among 9, 3, 4, and 11 which is 3. Because it is less than 12 (the value in element j = 0), it is swapped with 12, which produces the ordering shown in the second column of the table. Then the smallest value among 12, 4, and 11 is located, which is 4. Because it is less than **9** (the value in element j = 1), it is swapped with 9, producing the ordering shown in the third column of the table. The remaining two steps for j = 2 and j = 3 and the final sorted array are shown in the three right-most columns. When the array is an array of objects, the algorithm sorts the objects based on the value of one of the class's data members.

Table 6.1
The Progression of the Selection Sort Algorithm

	j = 0	j = 1	j = 2	j = 3	
index 0	12	3	3	3	3
1	9	9	4	4	4
2	**3**	12	12	9	9
3	4	**4**	**9**	12	11
4	11	11	11	**11**	12
	Original Array	After 12 and 3 were swapped	After 9 and 4 were swapped	After 12 and 9 were swapped	After 12 and 11 were swapped

The code shown on the left side of Figure 6.23 is an implementation of the Selection Sort algorithm for sorting an array of primitive values (in this case an array of people's ages). It uses nested loops: the inner loop searches for a minimum value, and the outer loop places it into its correct positioning in the array. The right side of the figure is an implementation of the algorithm used to sort an array of `ParentClass` objects based on the value of one of their primitive type data members (in this case, the integer data member `age`). Except for the names of the arrays, the type of the variable declared on line 2, and the use of the `getAge` method on lines 6, 10, and 12 on the right side of the figure, the implementations are identical.

The code that begins on line 6 and ends on line 15 is essentially the algorithm to locate the minimum value of an array's elements, which was discussed in Section 6.8.2 and implemented in Figure 6.22. The differences are that line 6 initializes the minimum value, `min`, to the jth element of the array rather than the first element, and line 7 initializes the minimum element number, `iMin`, to `j` rather than zero. In addition, the loop that begins on line 8 initializes its loop variable to `j + 1` rather than one. The variable `j` is the loop variable of an outer loop that begins on line 4 and ends on line 23. Because it is initialized to zero on line 4 the first time lines 6–15 execute, this code is in fact identical to the minimum value algorithm.

After the search for the minimum value ends (line 15), if `j` is not the index of the minimum value of the remaining unsorted portion of the array (as determined by the `if` statement on line 17), then lines 19–21 place the minimum value in element `j` by swapping element j with element `iMin`. The first time through the outer loop, element zero stores the minimum value contained in the array. After the second iteration of the outer loop, the next lowest element is stored in element 1. When the algorithm ends, the array is sorted in ascending order. To sort the elements of an array in *descending* order, the less than (<) operator in the Boolean condition on line 10 is changed to the greater than (>) operator.

It should be noted that when sorting an array of objects (right side of Figure 6,23), lines 19–21 swaps the references to the objects contained in the array parents. For example, on the first iteration of the outer loop, the location of the object whose `age` data member is the minimum value is placed in the first element of the array `parents`. The alternative is to swap the contents of the data members of the objects, which is more time consuming. In either case, if the sorted objects were output from the first element of the array to the last, they would appear in sorted order based on the contents of the `age` data member.

```
1   int iMin, min;
2   int temp;
3
4   for (int j = 0; j < ages.length; j++)
5   {
6      min = ages[j];
7      iMin = j;
8      for (int i = j+1; i < ages.length; i++)
9      {
10        if (ages[i] < min)
11        {
12            min = ages[i];
13            iMin = i;
14        }
15     }
16
17     if ( iMin != j )
18     {
19        temp = ages[j];
20        ages[j] = ages[iMin];
21        ages[iMin] = temp;
22     }
23  }
```

Sorting an Array of Primitives

```
1   int iMin, min;
2   ParentClass temp;
3
4   for (int j = 0; j < parent.length; j++)
5   {
6      min = parent[j].getAge();
7      iMin = j;
8      for (int i = j+1; i < parent.length; i++)
9      {
10        if (parent[i].getAge() <  min)
11        {
12             min = parent[i].getAge();
13            iMin = i;
14        }
15     }
16
17     if ( iMin != j )
18     {
19        temp = parent[j];
20        parent[j] = parent[iMin];
21        parent[iMin] = temp;
22     }
23   }
```

Sorting an Array of Objects

Figure 6.23
The Selection Sort algorithm.

The algorithm on the right side of Figure 6.23 can be used to sort an array of objects based on any primitive type data member contained in the array's objects, as long as the class of the objects contains a `get` method to fetch the data member (which would be invoked on lines 6, 10, and 12).

If the data member is *not* a primitive type variable, but rather a reference variable, then the class of the object it references also must contain a `compareTo` method to be used in the Boolean condition on line 10. Typically, this method returns a negative number when the object that invoked it is less than the argument sent to it (the variable `min`). The `String` class contains an implementation of this method. Assuming the data member referenced a `String` object and the data member's name was `lastName`, line 10 of the algorithm would become:

Line 10: `if(parent[i].getLastName().compareTo(min) <  0)`

When the array is an array of `String` objects, line 10 of the algorithm would become:

Line 10: `if(parent[i].compareTo(min) <  0)`

An Array Algorithm Case Study

Figure 6.24 presents the graphical application `ArrayAlgorithms` that illustrates the use of the array processing algorithms discussed in this chapter to process parallel arrays of snow families. When the application begins, the user is asked to enter the names of five snow families (e.g., “I”, “B”, “N”, “E”, and “G”). Then the five families, each consisting of a `ParentSnowman` and a `SnowChild` object, whose classes are shown in Figures 6.18 and 6.19, respectively, are displayed (Figure 6.25a).

After the program is launched, when the user types the name of a snow family (e.g., “I”), the family is searched for and alternately disappears (Figure 6.25b) and reappears (Figure 6.25a). When the up-arrow cursor key is struck, the family whose name is the minimum in sorted order (e.g., “B”), alternately disappears (Figure 6.25c) and reappears (Figure 6.25a). Finally, when the “S” key is struck, the visible snow families are displayed in sorted order by family name (Figure 6.25d).

```
import edu.sjcny.gpv1.*;
import java.awt.*;
import javax.swing.*;

public class ArrayAlgorithms extends DrawableAdapter
{
  static ArrayAlgorithms ge = new ArrayAlgorithms();
  static GameBoard gb = new GameBoard(ge, "ArrayAlgorithmsApp");
  static ParentSnowman[] parent;
  static SnowChild[] child;

  public static void main(String[] args)
  {
    String name;
```

```
    parent = new ParentSnowman[5];
    child = new SnowChild[5];

    for(int i = 0; i < 5; i++)
    {
      name = JOptionPane.showInputDialog("enter a family name");
      name = name.toUpperCase();
      child[i] = new SnowChild(50 + 60, 80 + 90 * i, name);
      parent[i] = new ParentSnowman(50, 50 + 90 * i, name);
    }
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 0; i<5; i++)
    {
      if(parent[i].getVisible() == true)
      { parent[i].show(g);
        child[i].show(g);
      }
    }
  }

  public void keyStruck(char key)
  {
    int index;

    String sKey = Character.toString(key);
    index = findValue(parent, sKey);
    if(index != -1) //name is valid, reverse family's visibility
    {
      if(parent[index].getVisible() == true)
      {
        parent[index].setVisible(false);
      }
      else
      {
        parent[index].setVisible(true);
      }
    }

    if (key == 'U') //up arrow struck, reverse visibility of min name
    {
      index = findMin(parent); //index of first family in alphabetic order
      if(parent[index].getVisible() == true)
      {
        parent[index].setVisible(false);
```

```java
      }
      else
      {
        parent[index].setVisible(true);
      }
    }
    if(key == 'S') //sort the families
    {
      selectionSort(parent);
    }
  }

  public static int findValue(ParentSnowman[] parent, String targetValue)
  {
    int elementNumber = -1;
    for(int i = 0; i< parent.length; i++)
    {
      if(parent[i].getName().equalsIgnoreCase(targetValue))
      {
        elementNumber = i;
        break;
      }
    }
    return elementNumber;
  }

  public static int findMin(ParentSnowman[] parent)
  {
    String min = parent[0].getName();
    int elementNumber = 0;
    for(int i = 1; i <  parent.length; i++)
    {
      if(parent[i].getName().compareToIgnoreCase(min) < 0)
      {
        min = parent[i].getName();
        elementNumber = i;
      }
    }
    return elementNumber;
  }

  public static void selectionSort(ParentSnowman[] parent)
  {
    int iMin, tempInt;
    ParentSnowman tempParent;
    SnowChild tempChild;
    String min;

```

```
111     for (int j = 0; j < parent.length; j++)
112     {
113       min = parent[j].getName();
114       iMin = j;
115       for (int i = j+1; i < parent.length; i++)
116       {
117         if (parent[i].getName().compareToIgnoreCase(min) <  0)
118         {
119           min = parent[i].getName();
120           iMin = i;
121         }
122       }
123       if(iMin != j) //swap element j with minimum element
124       {
125         tempParent = parent[j]; //swap array references
126         parent[j] = parent[iMin];
127         parent[iMin] = tempParent;
128         tempChild = child[j];
129         child[j] = child[iMin];
130         child[iMin] = tempChild;
131
132         tempInt = parent[j].getY(); //swap Y positions
133         parent[j].setY(parent[iMin].getY());
134         parent[iMin].setY(tempInt);
135         child[j].setY(parent[j].getY() + 30);
136         child[iMin].setY(parent[iMin].getY() + 30);
137       }
138     }
139   }
140 }
```

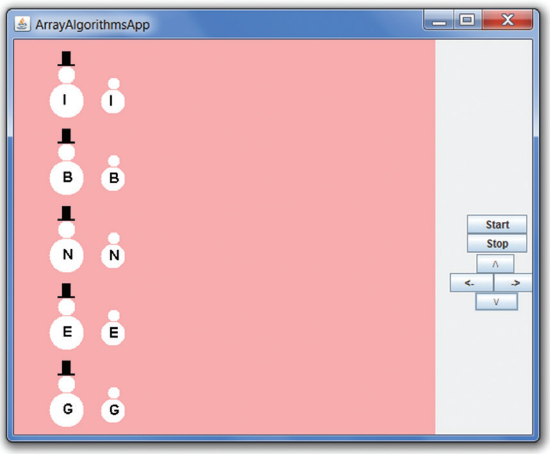
Figure 6.24
The application **`ArrayAlgorithms`**.

The loop that begins on line 18 of the `main` method accepts the five family names (line 20) and allocates the five child and parent objects (lines 22 and 23). With each pass though the loop, a child and a parent object are created, assigned positions next to each other on the game board, and given their family name. Then, the loop variable is used to write the addresses of these two newly created family members into the same (`i`th) element of the `child` and `parent` arrays, making these two arrays parallel.

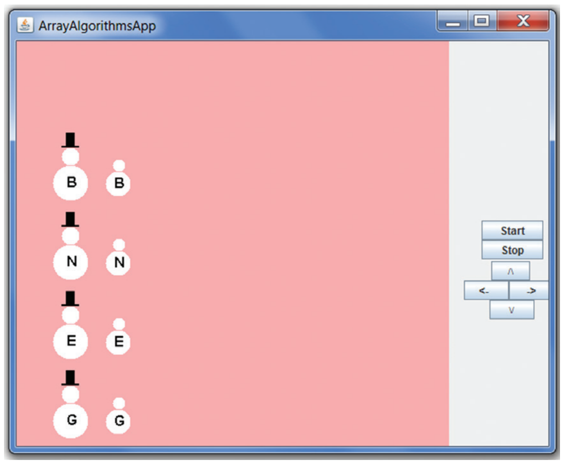
The loop that begins on line 30 of the `draw` method displays the five snow families on the game board, if they are visible as determined by the `if` statement's Boolean condition on line 32. When the Boolean condition is `true`, the `i`th parent and the `i`th child are drawn. Because the arrays were set up to be parallel, a parent and its child are drawn (lines 31–34).

Use of the Search Algorithm

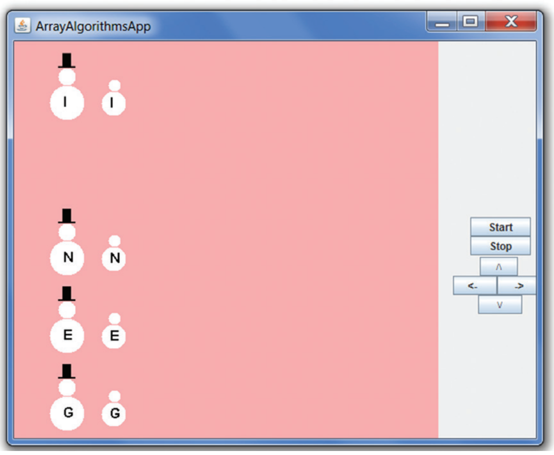
The code, to make a family alternately disappear and reappear when their one-character family name is typed, is on lines 43–55 of the `keyStruck` method. This method is invoked by the game environment whenever a key is typed, and the typed character is passed into the method's parameter, `key` (line 43). To locate a `ParentSnowman` object with that family name, line 44 invokes the `findValue` method (lines 75–87) passing it the parent array (`parent`) and the string version of the family name (`sKey`). The method `findValue` is an implementation of the object version of the search algorithm discussed at the end of Section 6.8.1. If an object is found with that family name (the returned array index is not -1 on line 45), then the index is used on lines 47–54 to reverse the visibility of the parent object. Assuming an "I" was struck twice, the game board would change from the board displayed in Figure 6.25a to that shown in Figure 6.25b and then back again.



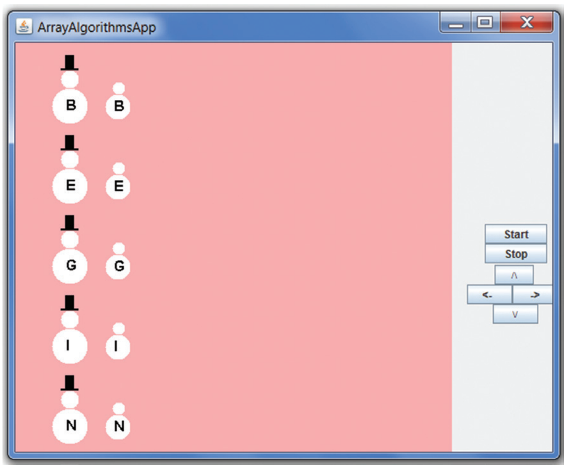
(a) After the program is launched



(b) After the "I" key is struck the first time



(c) Second strike of "I" key followed by a strike of the up cursor key



(d) After the "S" key is struck

Figure 6.25
The graphical output of the application **`ArrayAlgorithms`**.

Because in this application the search is for a particular value of a String object (the case-insensitive family name), this implementation of the search algorithm uses the String class's equalsIgnoreCase method on line 80 to compare the name passed to it (contained in the parameter targetValue) to the name returned by the ParentSnowman class's getName method.

Use of the Minimum Value Algorithm

The code to make a family whose name is first in alphabetical order alternately disappear and reappear when the up-arrow curser key is struck is coded on lines 57–68 of the keyStruck method. To locate a ParentSnowman object whose family name is first in alphabetic order, line 59 invokes the findMin method (lines 89–102), passing it the parent array. The method findMin is an implementation of the algorithm discussed in Section 6.8.2 that searches an array for a minimum value. In this coding of the algorithm, the method returns the array element number (line 101) that references the object whose name data member is first in alphabetic order (line 95). The returned element number is used on lines 60–67 to reverse the visibility of the parent object. Because the draw method (lines 28–37) only draws the snowman parent and its child when the parent's visible data member is true (line 32), both the parent and child disappear and reappear when the up-arrow key is typed.

In this application, the search is for a minimum value of a String object (the case-insensitive family name), findMin uses the String class's compareToIgnoreCase method on line 95 to determine if the name returned by the ParentSnowman class's getName method is less than the string referenced by min.

Use of the Selection Sort Algorithm

The code to sort the parent and child arrays in ascending order based on family names when the S key is struck is coded on lines 69–72 of the keyStruck method. To sort the parent and child arrays, line 71 invokes the selectionSort method (lines 104–139) passing it the parent array. The method selectionSort is an implementation of the sorting algorithm discussed in Section 6.8.3. In this coding of the algorithm, the method not only swaps the elements of the array passed to it (lines 125–127) but also the elements of its parallel array child (lines 128–130). (Because the address of the child array is declared as a class-level variable, the selectionSort method has access to it.) In addition, the method swaps the y data members of the parent objects (lines 132–134) and then positions the children next to their parents (lines 135–137), so the families will appear in sorted order from the top to the bottom of the game board.

Because in this application, the sorting is based on a String object (the case-insensitive family name), the method selectionSort uses the String class's compareToIgnoreCase method on line 117 to determine if the name returned by the ParentSnowman class's getName method is less than the string referenced by min.

6.9 APPLICATION PROGRAMMER INTERFACE ARRAY SUPPORT

The `System` and `Arrays` classes in the Java Application Programming Interface contain methods for processing arrays, and the API class `ArrayList` provides a means of storing data in an "array-like" object that can expand beyond its original size. The `ArrayList` class is one of the API's generic collection classes, which will be discussed in Chapter 13 after the topic of generics is introduced. We begin our discussion with the `System` class's `arraycopy` method.

6.9.1 The arraycopy Method

As its name implies, the `arraycopy` method in the `System` class is used to copy the contents of one array (called the *source* array) into another array (called the *destination* array). The method is a `static` method and is therefore invoked by first coding the name of its class, `System`, rather than the name of an object. Its signature contains five parameters, and a typical invocation would be:

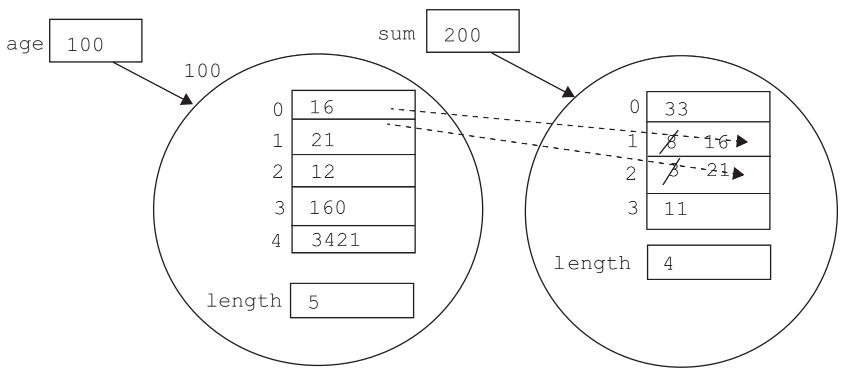
```
System.arraycopy(sourceArray, sourceIndex, destinationArray,
                            destinationIndex, numElements);
```

where:

```
sourceArray is the name of the array that is being copied (an array
            object reference variable)
sourceIndex is the index of the first element copied from sourceArray
            (an int)
destinationArray is the name of the array being copied to (an array
                 object reference variable)
destinationIndex is the index of the first element copied into
                 destinationArray (an int)
numElements is the number of sequential elements to be copied (an int)
```

Both the source array and destination arrays must exist (have been previously declared) before the method is invoked. Figure 6.24 shows the result of executing the invocation:

```
System.arraycopy(age, 0, sum, 1, 2);
```



`System.arraycopy(age, 0, sum, 1, 2);`

Figure 6.26
The use of the System class's **`arraycopy`** method.

If the `sourceIndex`, `destinationIndex`, and `numElements` values passed to the method are such that the copying causes element numbers to be generated that do not exist, a runtime error (ArrayIndexOutOfBoundsException) will occur.

6.9.2 The Arrays Class

The API `Arrays` class contains two methods that can be used to copy one array into another and contains several other useful methods for processing arrays of primitives and strings. These include a method to facilitate the output of all of the elements of an array by converting them to a single string, a sorting method, a search method, a method to determine if the corresponding elements of two arrays are equal, and a method that sets the elements of an array to a specified value. The searching and sorting method algorithms execute faster than the algorithms discussed in Sections 6.8.1 and 6.8.3. Some of the methods in the class `Arrays` are summarized in Table 6.2.

Table 6.2
Methods in the API Arrays Class

Function	Method Name	Typical Invocation
Copies all or a portion of an array beginning with the *first* element	`copyOf`	`int[] a = Arrays.copyOf(sourceArray, 10);` Returns an array containing the first *ten* elements of the array `sourceArray`
Copies an array beginning with *any* element	`copyRangeOf`	`int[] a = Arrays.copyRangeOf(sourceArray, 2, 10);` Returns an array containing the *third* (index 2) through the *ninth* elements of array `sourceArray`
Converts an array's contents to a string	`toString`	`String arrayContents = Arrays.toString(anArray);` Returns a string enclosed in braces containing the contents of the array's elements, each separated by a comma and a space
Sorts the elements of an array in ascending order	`sort`	Arrays.sort(anArray); Sorts all of the elements of the array `anArray` Arrays.sort(anArray, 1, 5); Sorts the values at index 1 through index 4 of the array `anArray`
Searches for (locates) a target value in a *sorted* array	`binarySearch`	`int i = Arrays.binarySearch(anArrray, targetValue)` Returns the index of an occurrence of `targetValue` in the sorted array anArray or returns a negative value if an occurrence is not found
Sets the elements of an array to a given value	`fill`	Arrays.fill(intArray, 4); Sets all of the elements of the array `intArray` to the value 4 `Arrays.fill(stringArray, 1, 4, "FillValue");` Sets the second (index 1) to the fourth (index 3) elements of the array `stringArray` to "FillValue"

All of these methods are static methods, and the program presented in Figure 6.27 illustrates their use. The output generated by this program appears in Figure 6.28.

Lines 15 and 16 of Figure 6.27 output all of the elements of the string and integer arrays `stringArray` and `intArray` created on lines 6 and 8. The parameter sent to the `println` method on lines 15 and 16 is the string returned from the `Arrays` class's `toString` method. The returned string is a concatenation of all of the elements of the array sent to the method separated by a comma and a space (lines 2 and 3 of Figure 6.28). It begins with an open bracket [and ends with a close bracket].

Lines 21 and 23 of Figure 6.27 use the `Arrays` class's static `copyOf` method to create a copy of the arrays `stringArray` and `intArray` and assign the newly created array addresses to the variables `copyStringArray` and `copyIntArray`, respectively. The second parameter sent to this method specifies the number of elements to be copied, and the copy always begins at element 0. If the number of elements to be copied exceeds the size of the source array (as it does on line 21), the elements are filled in with default values consistent with their type (`null` for string references) as shown on line 6 of Figure 6.28.

The `Arrays` class's `equals` method is used on lines 29 and 31 of Figure 6.27 to compare two integer arrays for equality. The method returns `true` if all of the corresponding elements of the two arrays passed to it are equal, as they are on line 29; otherwise, it returns `false` (line 31). The returned Boolean values are output and shown on lines 10 and 11 of Figure 6.28.

Line 36 of Figure 6.27 uses the one-parameter version of the `Arrays` class's static method `sort` to sort all of the elements of the array `stringArray`, and line 38 uses the three-parameter version of the method to sort the values at indices 1 through 4 of the array `intArray`. The third argument sent the three-parameter version of the method is always one larger than the index of the highest element to be sorted. This can be verified by comparing the output of the unsorted values (line 7 of Figure 6.28) to the output of the sorted values (line 15 of Figure 6.28). The sort is always performed inside the arrays passed to the method.

After the array `stringArray` is sorted (line 36), the `Arrays` class's method `binarySearch` can be used to determine the index of a given value in the array. Line 44 invokes this method to search the array for the index of the element containing the string "`Fred`." This string's position in sorted order is index 2 (line 14 of Figure 6.28), so the method returns the value 2 (line 18 of Figure 6.28). If there were several occurrences of the item being searched for in the array, it is uncertain as to which occurrence's index would be returned. Line 45 searches for the name "Doris," which is not contained in the array. When the item searched for is not in the array, a negative index is returned, in this case the value -3 (line 19 of Figure 6.28). When the search value is not found, the absolute value of the returned index is one greater than the index where the item would be if it were in its sorted position in the array. Line 14 of Figure 6.28 shows the sorted version of the array.

The `Arrays` class's methods `copyOf` and `copyRangeOf` can be used to copy all or part of the elements of an array. As previously stated, the `copyOf` method always begins its copy at index 0 of the array, and the second argument passed to it indicates the number of elements to copy. The method is used on line 50 of Figure 6.27 to copy the first four elements of the array `intArray` into a newly created array whose address is assigned to the variable `copyIntArray`. The contents of the returned array is shown on line 22 of Figure 6.28, which can be compared to the contents of the array `intArray` shown on line 15 of the figure.

When the `copyRangeOf` method is used, the copying can begin and end anywhere in the source array. As shown on line 52 of Figure 6.27, the source array is specified as the first argument sent to the method, the starting index is the second argument, and the third argument is always one more that the last index to be copied. Therefore, line 52 specifies that the elements at index 2–9 should be copied into a newly created array. If the value of the last argument specifies an index that is beyond the bounds of the source array (as it is on line 52), default values (e.g., zero for numeric types, `null` for `char` and `String` types) are entered into the out-of-bounds elements of the returned array. Thus, the last four elements in the integer array returned from the invocation on line 52 contain zeros (line 22 of Figure 6.28).

Lines 58 and 60 of Figure 6.27 use the `Arrays` class's `fill` method to set sequential elements of the arrays `intArray` and `stringArray` (specified as the first argument sent to this method) to a value specified by the last argument sent to the method. The two-parameter version of this method (invoked on line 58) fills all of the elements of the array sent to it. The four-parameter version of this method, invoked on line 60, fills a specified sequential range of elements of the array sent to it. The index at which to start the fill is the second argument sent to the four-parameter version of the method, and the third argument is always one more than the last index to be filled. The contents of the filled arrays are shown on lines 25 and 26 of Figure 6.28.

```
1    import java.util.Arrays;
2    public class ArraysClass
3    {
4      public static void main(String[] args)
5      {
6        String[] stringArray = {"Tom","Mary","Bob","Alice","Joe","Fred"};
7        String[] copyStringArray;
8        int[] intArray = { 3, 5, 2, 8, 6, 4};
9        int[] copyIntArray;
10       int[] filledIntArray;
11
12       //outputting the elements of an arrays
13       System.out.println("Outputing arrays using the " +
14                          "Arrays.toString method");
15       System.out.println(Arrays.toString(stringArray));
16       System.out.println(Arrays.toString(intArray));
17
18       //copying the elements of an array;
```

```
    System.out.println("\nCopying arrays using the " +
                       "Arrays.copyOf method");
    copyStringArray = Arrays.copyOf(stringArray, 10);
    System.out.println(Arrays.toString(copyStringArray));
    copyIntArray = Arrays.copyOf(intArray, intArray.length);
    System.out.println(Arrays.toString(copyIntArray));

    //determining if all the elements of two arrays are equal
    System.out.println("\nTesting two arrays for equality " +
                       "using the Arrays.equals method");
    System.out.println(Arrays.equals(intArray, copyIntArray));
    copyIntArray[0] = 1;
    System.out.println(Arrays.equals(intArray, copyIntArray));

    //sorting arrays
    System.out.println("\nSorting all or part of an array: " +
                       "the Arrays.sort method");
    Arrays.sort(stringArray);
    System.out.println(Arrays.toString(stringArray));
    Arrays.sort(intArray, 1, 5);
    System.out.println(Arrays.toString(intArray));

    //searching for an element of a sorted array
    System.out.println("\nSearching for a value: the " +
                       "Arrays.binarySearch method");
    System.out.println(Arrays.binarySearch(stringArray, "Fred"));
    System.out.println(Arrays.binarySearch(stringArray, "Doris"));

    //copying a part of an array
    System.out.println("\nPartial copies: the " +
                       "Arrays.copy and copyRange methods");
    copyIntArray = Arrays.copyOf(intArray, 4);
    System.out.println(Arrays.toString(copyIntArray));
    copyIntArray = Arrays.copyRangeOf(intArray, 2, 10);
    System.out.println(Arrays.toString(copyIntArray));

    //setting all elements of a array to one value
    System.out.println("\nFilling all or part of an array: " +
                       "the Arrays.fill method");
    Arrays.fill(intArray, 4);
    System.out.println(Arrays.toString(intArray));
    Arrays.fill(stringArray, 1, 4, "FillValue");
    System.out.println(Arrays.toString(stringArray));
  }
}
```

Figure 6.27
The application **ArraysClass**.

```
Outputting arrays using the Arrays.toString method
[Tom, Mary, Bob, Alice, Joe, Fred]
[3, 5, 2, 8, 6, 4]

Copying arrays using the Arrays.copyOf method
[Tom, Mary, Bob, Alice, Joe, Fred, null, null, null, null]
[3, 5, 2, 8, 6, 4]

Testing two arrays for equality: the Arrays.equals method
true
false

Sorting first all, then part of an array: the Arrays.sort method
[Alice, Bob, Fred, Joe, Mary, Tom]
[3, 2, 5, 6, 8, 4]

Searching for a value: the Arrays.binarySearch method
2
-3
Partial copies: the Arrays.copy and copyRangeOf methods
[3, 2, 5, 6]
[5, 6, 8, 4, 0, 0, 0, 0]

Filling all or part of an array: the Arrays.fill method
[4, 4, 4, 4, 4, 4]
[Alice, FillValue, FillValue, FillValue, Mary, Tom]

```

Figure 6.28
The output produced by the application **`ArraysClass`**.

6.10 Multi-dimensional Arrays

Java, like most programming languages, supports multidimensional arrays. Like one-dimensional arrays, each memory cell in a multidimensional array shares the same “first name,” the name of the array, and all of the array’s element must be the same type. To close out this analogy, unlike one-dimensional arrays, each element of multi-dimensional arrays have *two or more* unique last names.

The simplest multi-dimensional array is a two-dimensional array, which conceptually is a group of memory cells arranged in rows and columns (left side of Figure 6.29). In Java, there is no limit to the number of dimensions an array can have. Three-dimensional arrays can be visualized as multiple two-dimensional arrays each in a different plane (right side of Figure 6.30). In effect, we have added a depth dimension to the two-dimensional array. Because we live in a three-dimensional world, multi-dimensional arrays with more than three dimensions are not often used in programs, although there are times when they are useful.

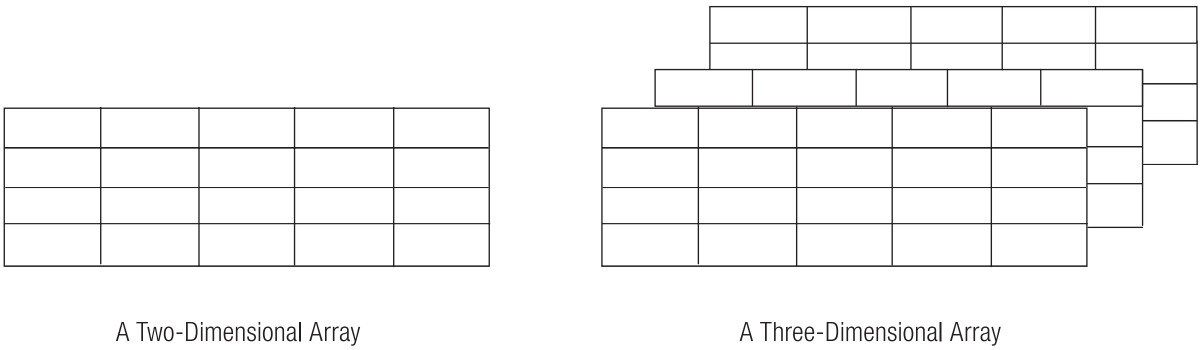


Figure 6.29
Visualization of a two- and three-dimensional array.

6.10.1 Two-Dimensional Arrays

A two-dimensional array is similar to a two-dimensional table with rows and columns. Two-dimensional arrays are typically used to store a group of data items for several entities. For example, a group of 5 examination grades for each of 4 students, or the 10 qualifying times for each of the 100 cyclists in the Tour de France. (A three-dimensional array with five planes could store the last five years' Tour de France qualifying time results, one year per plane.)

The names of the memory cells (elements) that make up a two-dimensional array begin with the name of the array followed by the element's row and column number. Figure 6.30 shows the names of all of the twenty elements of a four-row by five-column two-dimensional array named `grades`.

The name of an array element can be used anywhere the name of a non-array variable can be used, e.g., in arithmetic expressions, in output and input statements, in argument lists, and on the right side of the assignment operator. When it is used, the element's row number is always written *before* its column number. Thus, we would code `grades[2][4]` to access the contents of the third row and fifth column of the `grades` array shown in Figure 6.30.

NOTE *The indices of both the row and column numbers start from zero.*

	column 0	column 1	column 2	column 3	column 4
row 0	grades[0][0]	grades[0][1]	grades[0][2]	grades[0][3]	grades[0][4]
row 1	grades[1][0]	grades[1][1]	grades[1][2]	grades[1][3]	grades[1][4]
row 2	grades[2][0]	grades[2][1]	grades[2][2]	grades[2][3]	grades[2][4]
row 3	grades[3][0]	grades[3][1]	grades[3][2]	grades[3][3]	grades[3][4]

Figure 6.30
The element names of a four-by-five two-dimensional array named grades.

The syntax used to declare a two-dimensional array is a simple extension of the one-dimensional array declaration syntax discussed in Section 6.3. The only difference is that an additional set of empty brackets is added before the name of the array, and the number of columns is specified after the number of rows. (In a similar way, a third set of empty brackets and the number of planes is added to declare a three-dimensional array.) The Java statement to declare the two-dimensional *integer* array `grades`, shown in Figure 6.30, would be:

```
int[][] grades = new int[4][5]; //four rows and five columns
```

The equivalent two-line syntax is:

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

The numeric literals in the second line of the two-line grammar can be replaced with integer variables to specify or change the size of the array at run time.

As is the case with all array declarations, these declarations create a reference variable named `grades` that refers to an array object containing the array's elements. To facilitate the processing of all of the elements of a two-dimensional array, the array object also contains one additional `public` data member *per row* that stores the number of elements (columns) in each row of the array. For the array `grades`, depicted in Figure 6.30, each of these variables would store the integer value 5, and their names would be:

`grades[0].length`, `grades[1].length`, `grades[2].length`, and `grades[3].length`

The array object would still contain the `public` data member `grades.length` that stores the number of rows contained in the array (the integer value 4).

Initializing Two-Dimensional Arrays

The elements of a two-dimensional array can be initialized when the array is declared. As with one-dimensional arrays, when this is done the number of rows and columns in the array is not specifically stated but is implied from the number of initial values, and an initial value must be specified for each element of the array. The initialization syntax is most easily understood if we consider a two-dimensional array to be a one-dimensional array with each of its elements being a row of the two-dimensional array. In addition, it is most easily read if we write each row's initial values on a separate line. The array declaration below uses this coding style to declare the two-dimensional array `ages` depicted in Figure 6.31, initializing all of its values to those shown in the figure.

```
int[][] ages = { {10, 11, 12, 13, 14},
                 {20, 21, 22, 23, 24},
                 {30, 31, 32, 33, 34},
                 {40, 41, 42, 43, 44} };
```

NOTE *If one element of an array is to be initialized, all of the elements must be initialized.*

	column 0	column 1	column 2	column 3	column 4
row 0	10	11	12	13	14
row 1	20	21	22	23	24
row 2	30	31	32	33	34
row 4	40	41	42	43	44

Figure 6.31
The elements of the array grades after its initialization.

The number of initial values specified in each row of a two-dimensional array need not be the same. The following declaration produces the array `ages`, shown in Figure 6.32, which has a different number of elements in each of its three rows:

```
int[][] ages  = { {10, 11, 12},
                  {20, 21, 22, 23, 24},
                  {30, 31} };
```

Inside the array object `ages`, the `public` variables named `ages[0].length`, `ages[1].length`, and `ages[2].length` would store the integer values 3, 5, and 2, respectively. These variables are used in the code fragment below to output all of the elements of the array `ages` shown in Figure 6.32 and are often used by methods that are passed two-dimensional arrays to determine the number of columns in each row of the array.

```
for(int row = 0; row < ages.length; row++) //each row
{
    for(int col = 0, col < ages[row].length; col++) //each column in a row
    {
        System.out.print(ages[row][col] + " ");
    }
    System.out.println();
}
```

	column 0	column 1	column 2	column 3	column 4
row 0	10	11	12		
row 1	20	21	22	23	24
row 2	30	31			

Figure 6.32
The elements of the array ages after its initialization.

6.11 DELETING, MODIFYING, AND ADDING DISK FILE ITEMS

In Section 4.8.5, it was mentioned that Java, like most programming languages, does not contain a method to delete or modify a file data item or to add a data item anywhere in the file except at its end. These operations can be performed by algorithms that *combine* the disk I/O methods discussed in Chapter 4 with the use of arrays and loops. An array is used because *all* of the data must be read into RAM memory to delete or modify a data item or to add an item to an arbitrary position in a data file. Because disk data files normally contain large data sets, this can easily be accomplished by reading each item into an element of the array and placing the read statement inside a loop.

In the remainder of this chapter, we will discuss these algorithms and their processing of a file of primitive data items that are all of the same type. The use of algorithms to process a file that contains the data members of several objects, with the data members possibly being different types, will be discussed in Chapter 7.

Deleting an Item From a Disk File

The algorithm to delete a data item from a file would be:

1. Open the file, read the number of items contained in the file, and allocate an array of that size
2. Inside a `for` loop, read all of the file's data items into the array
3. Close the file
4. Delete and recreate the file and write the new number of items to the file
5. Inside a `for` loop, write the elements of the array, *except* the item to be deleted, to the file
6. Close the file

When the algorithm ends, there would be one less item in the file, the deleted item, and the remaining data would be in their original order.

The item to be deleted can be specified by its position in the file, e.g., "delete item number 25," or by specifying the data value to be deleted, e.g., "delete the deposit 34.56." When the position in the file is specified, Step 5's loop variable is used to decide if the next element of the array should be written to the file. Assuming Step 1 of the algorithm stored the number of items read from the file in the variable `count`, and Step 2 reads the data into the array `data`, the following code fragment illustrates Step 5's process when deleting item 25 from the file:

```
//delete item 25
for(int i = 0; i < count; i++)
{
  if(i != 25 - 1)
  {
    //write the item, data[i], to the file
  }
}
```

When the value of the data item to be deleted is specified, the decision to write the next element of the array back into the file is based on the contents of the array element. Assuming Step 1 of the algorithm stored the number of items read from the file in the variable `count` and then read the data into the array `data`, the following code fragment illustrates Step 5's process when deleting all of the occurrences of 35.56 from the file:

```
//delete 35.56 from the file
for(int i = 0; i < count; i++)
{
  if(data[i] != 35.56)
  {
    //write the item, data[i], to the file
  }
}
```

The application `DeleteFileItem` shown in Figure 6.33 deletes a game score input by the program's user from the disk text file `scores.txt`. It uses the six-step algorithm discussed in this section and the file input and output methods discussed in Chapter 4. The number of items in the file is read from the file on line 16 and then used to size the array on line 17. It is also used on line 31 to write the new number of file items into the file after it is deleted and recreated by lines 29 and 30. The value to be deleted from the file is parsed into the variable `deletedItem` on line 35 and then used in the Boolean condition on line 38 to prevent the deleted item from being rewritten to the file.

```
1    import java.util.Scanner;
2    import java.io.*;
3    import javax.swing.*;
4
5    public class DeleteFileItem
6    {
7      public static void main(String[] args) throws IOException
8      {
9        double[] data;
10       double deletedItem;
11       int count = 0;
12
13       //Step 1: Open the file, read the number of items, allocate the array
14       File fileObject = new File("score.txt");
15       Scanner fileIn = new Scanner(fileObject);
16       count = fileIn.nextInt();
17       data = new double[count];
18
19       //Step 2: Read all of the file's data items into the array
20       for(int i = 0; i < count; i++)
21       {
22         data[i] = fileIn.nextDouble();
23       }
24
25       //Step 3: Close the file
26       fileIn.close();
27
```

```
28        //Step 4: Delete and recreate the file
29        FileWriter fileWriterObject = new FileWriter("data.txt");
30        PrintWriter fileOut = new PrintWriter(fileWriterObject, false);
31        fileOut.println(count - 1);
32
33        //Step 5: write the elements of the array without the deleted item
34        String s = JOptionPane.showInputDialog("enter score to delete");
35        deletedItem = Double.parseDouble(s);
36        for(int i = 0; i < count; i++)
37        {
38          if(data[i] != deletedItem)
39          {
40            fileOut.println(data[i]);
41          }
42        }
43
44        //Step 6: Close the file
45        fileOut.close();
46      }
47    }
```

Figure 6.33
The application **DeleteFileItem**.

Detecting an End of a File

If the number of items in the file was not stored in the file, then line 16 of Figure 6.33 would be replaced with the following code fragment that counts the number of items in the file before the array is declared and then closes and reopens the file. In addition, line 31 would be removed from the program.

```
while(fileIn.hasNext()) //count the data items
{
  count++;
  fileIn.nextDouble();
}
fileIn.close();
fileObject = new File("data.txt");
fileIn = new Scanner(fileObject);
```

Modifying an Item Stored in a Disk File

The algorithm to modify an item in a disk file would be the same as the deletion algorithm except that an `else` clause would be added to Step 5's `if` statement (line 38 of Figure 6.33). Assuming the new value of the data item was stored in the variable `newValue`, the `else` clause would be coded as:

```
else
{
  //write the new value to the file
}
```

Because the number of data items in the file would remain the same, Step 4 (line 31 of Figure 6.33) would write the original number of data items back into the file.

Inserting a New Item into a File

To insert a new item into a file, its position in the file and its value must be known. The algorithm, shown below, is the same as the deletion algorithm except for Step 5, and the number of items in the file written to the file in Step 4 would be increased by one:

1. Open the file, read the number of items contained in the file, and allocate an array of that size
2. Inside a `for` loop, read all of the file's data items into the array
3. Close the file
4. Delete and recreate the file and write the new number of items to the file
5. Inside a `for` loop, write the elements of the array, *and the new item* to the file
6. Close the file

Assuming the following: the new item's position in the file is stored in the variable `itemNumber`; the new value is stored in the variable `newValue`; and position numbers in the file begin at zero, then the Step 5 `for` loop becomes:

```
//add newValue to the file at position itemNumber
for(int i = 0; i < count; i++)
{
  if(i == itemNumber)
  {
    //write the newValue to the file
  }
  //write the item, data[i], to the file
}
```

In addition to modifying text files and data files, these file operations will also enable us to record players' scores and update the high scores of a game.

6.12 CHAPTER SUMMARY

The concept of an array presented in this chapter provides us with a powerful tool for storing, retrieving, and processing data, especially large data sets. Unlike primitive variables, which contain only a single value, an array contains multiple data elements that are all of the same type. When an array is created, its data items are all initialized to their default values. The index of an array always begins at zero and extends to n-1, where n is the number of elements in the array (the *size* of the array). To distinguish one element from another, we use an index after the array name, as in an element's name, `grade[4]`, which is the name of the fifth element of the array.

In Java, all arrays are stored inside an object. This object is declared using a syntax similar to that used to declare non-array objects. A set of brackets is added to the declaration after the object's type, and a second set of brackets containing the size of the array is written after the keyword `new`.

Every array has a public data member named `length`, which stores the size of a one-dimensional array or the number rows in a two-dimensional array. Two-dimensional arrays also contain an array of public data members named `length` whose elements store the number of elements in each row of the array.

Loops are often used to efficiently input, output, and process the data in an array using the loop variable as an index into the array. Array elements can be used in any Java statement where a variable can be used; they can receive input and be output, used in mathematical expressions, assigned values, passed as an argument into a method, and returned from a method. In addition, like any object, the location of an array object can be passed to a method and returned from it. The concept of parallel arrays is implemented using either multiple one-dimensional arrays or using multidimensional arrays. In either case, we can use this concept to organize related information such as student ID numbers and GPAs and quickly and efficiently access it.

Sorting and searching for particular values, including minimum and maximum values, are very common programming operations, and their algorithms are very similar. The Selection Sort algorithm can be used to sort an array in ascending or descending order. It uses nested loops to locate the smallest (or largest) value in the array and to store it in the first element. Then, it searches for the next smallest value and stores it in the second element, and repeats this process until the entire list is sorted.

An array's elements can be either primitive or reference variables. An array of objects can be simulated by creating an array in which each element is a reference variable. For example, we can create an array of snowmen, or more correctly, an array of reference variables that refer to `Snowman` objects. The position or speed of all these `Snowmen` objects can easily be changed within a loop to create a parade of `Snowmen` objects that appear to be marching around the game board screen.

The `System` class method `arraycopy` is used to copy a sequential set of elements from one array into another array. Other API methods include `toString`, `sort`, and `fill`, which converts all of an array's elements to a single string, efficiently sorts the elements into ascending order, or sets an array's elements to a given value, respectively.

Finally, arrays are used in the implementation of algorithms that insert new items into a text file and that delete or update existing items. These algorithms use the disk I/O methods discussed in Chapter 4 to read an existing data set from a disk file into an array. Then, the file is deleted and recreated, and the data set is written back to the file with new items inserted into it, or existing items deleted or updated.

Knowledge Exercises

1. True or false:
- **(a)** An array is a technique for naming groups of memory cells.
- **(b)** All the elements of an array need *not* be of the same data type.
- **(c)** The size of an array can be dynamically allocated.

(d) Once an array is created, its size cannot be changed.
(e) The largest index of an array is its length - 1.
(f) An array cannot be passed into a method.
(g) Arrays can only be one or two dimensional.
(h) Arrays can be initialized when they are created.
(i) An efficient way to perform an operation on an entire array is to process it in a loop.
(j) In Java, the first index or subscript always begins with 1.

2. Give two features of an array that makes it more powerful than a set of non-array variables.

3. Mention at least two differences between an array element and a non-array variable.

4. Explain the difference between an array and an array object.

5. Assume that the array `gameScores` has been created using the following declaration:

```
int[] gameScores = new int[100];
```

Answer the following questions with respect to this array:

True or false:
(a) The size of this array is 100.
(b) The first element in this array is `gameScores[1]`
(c) The last element in this array is `gameScores[100]`
(d) This is a valid assignment to this array: `gameScores[5] = 93.2;`
(e) When invoked, `gameScores.length` would return the value 99.
(f) Give the Java statement to store the value 12 the second element of the array.
(g) Give the statement to output the last element of the array to the system console.
(h) Give the Java statements to output all of the elements of the array to the system console.
(i) Give the Java statements to output the average of all of the elements of the array.
(j) Draw a picture of the memory allocated by the declaration.

6. Find at least two errors in the following code and explain what should be done to fix them:

```
//prices start at $0 (not available) and increase from $5, to $10, $15….
    int size = 25;
    double[] ticketPrice = new double [size];
    for (int i = 1; i <= 25; i++)
    {
      ticketPrice[i] = i * 5;
    }
    System.out.println ("The price of a tier " + " i " + " ticket is:  " +
                                  ticketPrice[i]);
```

7. Assume that you have been given these declarations below, answer the questions that follow them.

```
final int MAX = 45;
int[] x = new int[MAX];
double[] y = {22.54, 3.6, 54.76, 10.8, 5.62};
double z;
```

(a) How many elements does the array `x` contain?
(b) What is the subscript or index of the last element of array `x`?
(c) What is the largest valid subscript of array `y`?
(d) Write the statements to multiply the very first element of the array by 7.
(e) Write the statements that increase the last element of array `y` by 20.5.
(f) Write a statement that assigns the sum of the first 3 elements of array `y` to `z`.

8. Give a statement to allocate an array that can store:
(a) Three thousand characters using the one-line declaration syntax
(b) Two hundred strings using the two-line declaration syntax
(c) Five thousand `Snowmen` objects using the one-line declaration syntax
(d) Five quiz scores for 100 students

9. Give the statement(s) to:
(a) Declare three parallel arrays that can store the names, weights, and target weights for 50 people in a weight loss clinic
(b) Output all of the information stored in the arrays declared in part a to the system console, one person per line
(c) Output Joe Smith's weight and target weight to the system console
(d) Output all of the names to the system console in alphabetical order

10. Write a method that is passed two arrays of doubles, each of the same size, and returns an array whose ith element is the sum up to the ith elements of the two arrays passed to it.

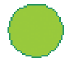
Programming Exercises

1. Refer to the program in Figure 6.24. Modify the code in the `keyStruck` method to include the instructions to move the third parent snowman and its child 20 pixels to the right each time the "M" key is struck.

2. As part of a research project, you have collected the following data and have initialized and stored it in an initialized array using the declaration:
```
int[ ] ages = {21, 32, 45, 23, 19, 41, 27, 20 , 21, 43,
               39, 24, 25, 22, 44};
```
Write a program to do each of the following tasks (with all output going to the system console):
(a) Search for a value input by the user and report if it is found or not
(b) Search for and output the minimum and maximum ages in the data set
(c) Calculate and output the average age
(d) Copy the ages to a new array called `sortedAges`, sort this array in ascending order, and then output both arrays

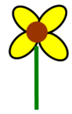
3. Write a program to accept a given number of names, input by the program user, and write the names in sorted order to a disk file named Students.txt. Then, ask the user which name should be eliminated from the file and eliminate that name from the file. Finally, read all of the names from the modified file and output them in reverse alphabetical order.

4. Write a graphical application that includes a class that defines a solid disk object whose diameter, color, and location are specified when an object is constructed. When the program is launched, the user should be asked how many disks to display on the game board, then asked the size, color, and location of each disk. The disks should then be displayed on the game board at their specified locations.



5. Modify the program described in Programming Exercise 4 so only disks whose diameters are 50 pixels or smaller are displayed when the down-arrow key is struck and only those disks that are larger than 50 pixels are displayed when the up-arrow key is struck.

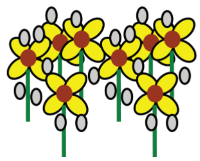
6. Write a graphical application that includes a class that defines a flower with a red center whose petal color and location is specified when a `Flower` object is constructed. When launched, the program should display a garden of 100 flowers at random locations on the lower portion of the game board. The (x, y) locations of the flowers will be randomly generated and be in the range ($7 \le x \le 500 - w$) and ($300 \le y \le 500 - h$), where w and h are the width and height of the flower.



7. Modify the program described in Programming Exercise 6 so every time the down-arrow game board button is clicked, 20 of the remaining flowers disappear from the garden.

8. Write a graphical application that includes a class that defines a light-gray colored raindrop whose height is 4 pixels and whose width is 6 pixels. When the application is launched, 300 raindrops should appear on the game board at random locations. The (x, y) locations of the raindrops will be in the range ($7 \le x \le 496$) and ($30 \le y \le 494$).

9. Modify the program described in Programming Exercise 8 so when the Start button on the game board is clicked, the raindrops move downward two pixels every 40 milliseconds, giving the appearance that it is raining. When a raindrop reaches the bottom of the game board (the y coordinate of its location is greater than 500), reset its y coordinate to 30.



10. Add the garden described in Programming Exercise 6 to Programming Exercise 8.

11. Add the garden described in Programming Exercise 6 to Programming Exercise 9.

12. Using the skills developed in this chapter, continue the implementation of the parts of your game that require multiple instances of one of your game pieces. To facilitate the processing of these objects, they should be part of an array of objects.

Enrichment

Investigate at least two other sorting algorithms and discuss their advantages and disadvantages over the Selection Sort algorithm.

Investigate why the `binarySearch` method in the API `Arrays` class is faster than the search method presented in this chapter.

References

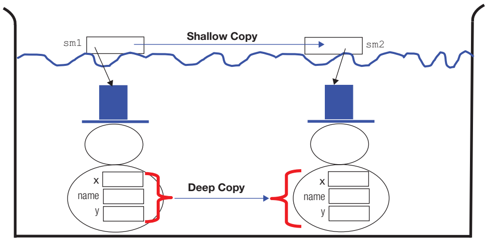
Knuth, Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching,* 2nd ed. New York: Addison-Wesley, 1998.

Levitin, Analy. *Introduction to the Design and Analysis of Algorithms*, 3rd ed. New York: Addison-Wesley Longman, 2011.

McAllister, William. *Data Structures and Algorithms*. Sudbury, MA: Jones and Bartlett Publishers, 2009.

CHAPTER 7

METHODS, CLASSES, AND OBJECTS: A SECOND LOOK



In this chapter

In this chapter, we will extend our knowledge of the features that can be incorporated into the classes we write, our knowledge of the `String` and the wrapper classes, and explore two other often-used classes defined in the Java Application Programming Interface (API). We will learn the techniques and motivation for writing classes whose objects share a data member and whose methods invoke each other, as well as the techniques and motivation for defining classes whose data members are objects. In addition, we will discuss what it means to compare, copy, and clone objects and how to write methods that perform these operations. An understanding of the topics in this chapter will enable us to more efficiently write complex programs, increase the reusability of the classes we write, and process numeric values that are beyond the size and precision of the primitive numeric types.

After successfully completing this chapter, you should:

- Understand static data members and their ability to share storage among all instances of a class within an application
- Become more familiar with the distinction between public and private methods
- Understand the fundamental differences between deep and shallow comparisons and copies
- Be able to compare two instances of a class
- Be able to use the deep copy technique to copy data members and clone objects

- Know how to create strings from primitive values, convert strings to characters, tokenize a string, and utilize other common string-processing methods
- Be able to create and use a wrapper class object and its autoboxing feature
- Understand how to aggregate an object into a class and the advantages of doing so
- Comprehend the relationship between, and implementation of, inner and outer classes
- Be able to use the `BigInteger` and `BigDecimal` classes for processing large numbers of arbitrary precision

7.1 STATIC DATA MEMBERS

Consider a worker class named `Student` that contains two data members named `idNumber` and `gpa` and a two-parameter constructor. The following code fragment would produce the `Student` objects `ryan` and `mary` depicted in Figure 7.1, assuming the first and second arguments passed to the constructor are used to initialize data members `idNumber` and `gpa`, respectively. As discussed in Chapter 3, each instance contains storage for its two data members.

```
Student ryan = new Student(1567, 3.26);
Student mary = new Student(2373, 2.87);
```

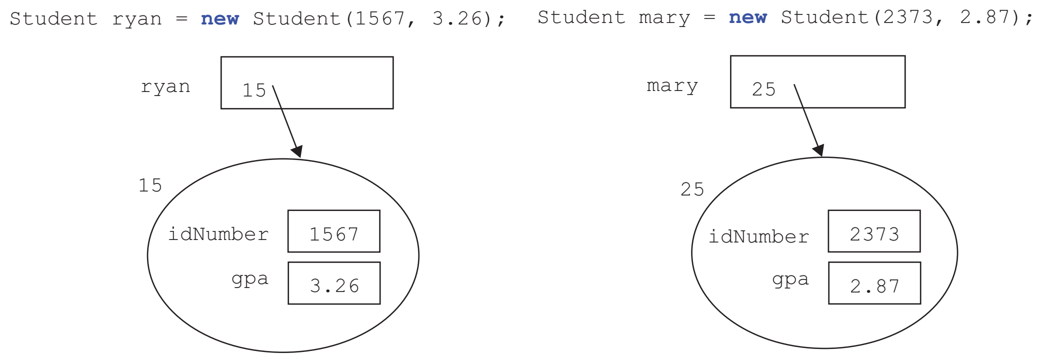


Figure 7.1
Two **`Student`** objects and the data members allocated to them.

Like methods, data members of a worker class can be declared to be static data members by including the keyword **`static`** in the data member's declaration statement. For example:

```
private static int studentCount = 0;
```

When a data member of a class is declared to be static, each instance of the class declared within an application does not contain storage for the data member. Rather, *one* storage cell is *shared* among all objects declared within an application. For example, if an additional static data member named `studentCount` were added to the class `Student`, the memory allocated to the two `Student` instances `ryan` and `mary` shown in Figure 7.1 would be expanded by one shared integer variable shown at the bottom of Figure 7.2.

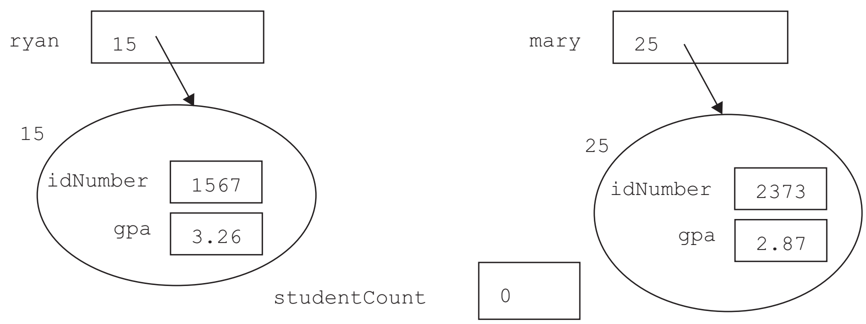


Figure 7.2
Two **Student** objects and the shared static data member **studentCount**.

A very common use of static data members is to keep track of the number of instances of a class (objects) that have been declared within an application. To accomplish this, a line of code to increment the static data member is included in each of the class's constructors. Below is an implementation of the `Student` class's two-parameter constructor that uses the class's static data member `studentCount` to keep track of the number of objects declared in the class:

```
public Student(int idNumber, double gpa)
{
  studentCount++; //counts the number of Student objects declared
  this.idNumber = idNumber;
  this.gpa = gpa;
}
```

Figure 7.3 shows the changes to the data member `studentCount` after two `Student` objects have been constructed with this version of the constructor. Normally, static data members in a class are declared with *private* access, and a `get` method is coded in the class to fetch the value of the data member. When the data member is being used to count the instances of a class, a `set` method is not coded to generally limit the data member's write access to the class's methods (and specifically, to the class's constructors).

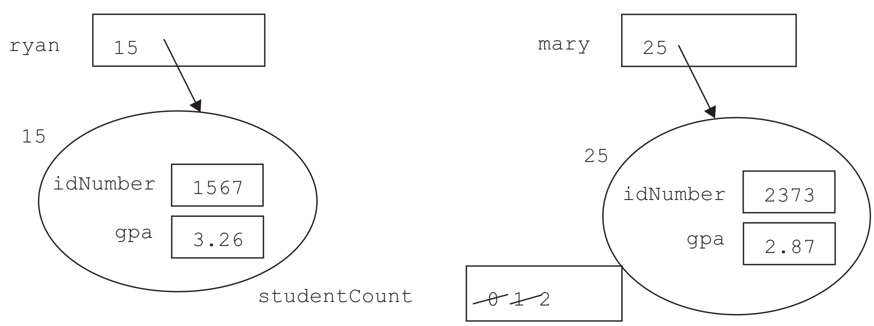


Figure 7.3
The changes to the static data member **studentCount** after two objects are created.

It is good coding style to declare the `get` method to be a static method. This forces the invoker (as shown below) to code the name of the class in the invocation statement rather than the name of an object, which implies to its reader that the value being fetched does not belong to a particular object. For example:

```
int numberOfStudents = Student.getStudentCount();
```

Figure 7.4 uses the concepts discussed in this section to keep track of the number of instances of the class `Student` that are declared within in a program. Line 2 declares the static data member `studentCount` and initializes it to zero. This variable is incremented within the class's two-parameter constructor on line 8 every time the constructor is invoked to create an object. The class's `toString` method (lines 13–18) does not return the value of the static variable because normally the string it returns includes only the values of a particular object's data members. The number of `Student` instances declared in a program can be fetched by invoking the class's `getStudentCount` method (lines 20–23). As previously discussed, this is a static method, and the class does not contain a `setStudentCount` method to restrict applications' write access to the static variable.

```
1   public class Student
2   { private static int studentCount = 0;
3     private int idNumber;
4     private double gpa;
5
6     public Student(int idNumber, double gpa)
7     {
8       studentCount++; //counts the number of Student objects declared
9       this.idNumber = idNumber;
10      this.gpa = gpa;
11    }
12
13    public String toString()
14    {
15      String s = "id is " + idNumber +
16                 "\ngpa is "  + gpa;
17      return s;
18    }
19
20    public static int getStudentCount()
21    {
22      return studentCount;
23    }
24  }
```

Figure 7.4
The worker class **`Student`**.

7.2 METHODS INVOKING METHODS WITHIN THEIR CLASS

A method in a worker class can invoke another method in its class. This is a common coding technique and, if used properly, can reduce the time required to develop a class and make our programs easier to read and understand.

Suppose that the UML diagram that specified the class `Student` shown in Figure 7.4 also required a method named `show` to be part of the class that outputs the annotated values of the data members of an object to the system console. One way to code the method would be:

```
public void show()
{
  String s = "id is " + idNumber +
             "\ngpa is "  + gpa;
  System.out.println(s);
}
```

However, a better way to code this method would be to take advantage of the fact that the UML diagram also specified that a `toString` method would be part of the class, and a method in a class can invoke other methods in its class. Knowing this, the `show` method would be coded after the `toString` method was completed and verified (taken for a test drive), so it could be invoked to perform some work for the `show` method. This approach reduces the code of the `show` method to one executable statement, as shown below:

```
public void show()
{
  System.out.println(toString()); //toString does all the work
}
```

Figure 7.5 is an expanded version of the class `Student` shown in Figure 7.4 with this coding of the `show` method added to it (lines 25–28).

Normally, when a nonstatic worker method is invoked within client code, its name is preceded by the name of an object followed by a dot. It would be impossible to use this syntax to invoke the `toString` method on line 27 of Figure 7.5 because objects (instances of worker classes) are declared in client code. Because line 27 is syntactically correct, the question of which object's data members will be output to the console by the invocation of `toString` on line 27 arises. The answer is: the object the client code used to invoke the `show` method. When a nonstatic method of a class is invoked by another method in the class, the method operates on the same object upon which the method invoking it is operating.

For example, the `toString` method invoked on line 27 of Figure 7.5 would return a string containing Ryan's student information when the following client-code fragment was executed:

```
StudentV2 ryan = new StudentV2(1567, 3.26);
ryan.show();
```

NOTE

When a method in a class invokes another method in its class, the invocation statement is not preceded by the name of an object followed by a dot, and both methods operate on the same object.

```
public class StudentV2
{ private static int studentCount = 0;
  private int idNumber;
  private double gpa;

  public StudentV2(int idNumber, double gpa)
  {
    studentCount++; //counts the number of Student objects declared
    this.idNumber = idNumber;
    this.gpa = gpa;
  }

  public String toString()
  {
    String s = "id is " + idNumber +
               "\ngpa is "  + gpa;
    return s;
  }

  public static int getStudentCount()
  {
    return studentCount;
  }

  public void show()
  {
    System.out.println(toString());
  }
}
```

Figure 7.5
The class **StudentV2**.

Private Class Methods

Another example of a worker class method invoking a method in its class evolves from the design process discussed in Section 1.7. When a UML diagram of a class specifies that a complicated method is to be included in the class, it is good programming practice to divide it into several simpler methods that are added to the complicated method's class. This is consistent with the divide and conquer problem-solving technique. Once each of the simpler methods have been coded and tested, often in parallel by several different programmers, the complicated method is written as a series of invocations of the simpler methods that are part of its class.

Because the only reason the simple methods were written was to perform the work of a more complex method in their class, the simpler methods are normally declared to be *private* methods. Private methods can only be invoked by the code of other methods within their class, and their signature begins with the keyword **`private`** rather than with the keyword **`public`**. When this is done, we say that the method has private access, and an attempt to invoke a private method from within a method that is not a member of its class results in a translation error. Often, methods are declared private to prevent methods that are not part of their class from invoking them. We will discuss this further in Section 7.4.

7.3 COMPARING OBJECTS

In Section 6.8, we discussed algorithms for searching, finding minimums and maximums, and sorting an array of objects. Fundamental to all of these algorithms is the ability to compare two objects. Generally speaking, the phrase "compare two objects" is ambiguous. It could mean that we want to compare the contents of a particular data member of two objects or the contents of two or more data members of two objects, or it could mean that we want to compare the contents of the reference variables that refer to the objects. Therefore, before we write a method that compares two objects for a particular application, we have to define what it means to "compare two objects" in the context of that application.

The simplest case is when the objects being compared are strings, but even then we would have to decide whether to simply compare the length of the strings or compare the strings for equality or lexicographical order and decide if these comparisons should be case sensitive. Once we define what it means to compare two string objects for a particular application, then in most cases, either the `String` class's length method or its `equals` or `compareTo` method (or its case-insensitive versions of these methods) can be used to compare the strings. The use of these methods to compare string objects was discussed in Section 4.2.3, and the use of these methods to compare data members of objects that are strings was discussed in Section 6.8.

When the objects being compared are not strings, we normally add a method to the object's class to perform the comparison after defining what it means to compare two objects. It is good coding style to name these methods `equals` or `compareTo`, or to at least use these words in a longer method title.

A fish tank analogy is useful in gaining an understanding of how to write and use these methods as well as the methods that will be discussed in the next section of this chapter. Figure 7.6 depicts two snowman objects: `sm1` and `sm2` in a fish tank. In this analogy, reference variables float at the top of the tank because they are light (They only contain one address). Objects are depicted at the deeper levels of the tank because they contain multiple data members, which make them heavy. Our snowmen contain three data members: each snowman's (x, y) location and its reference variable name.

A *shallow* comparison is performed at the surface of the tank. It compares the contents of the two reference variables that float on the surface of the tank (e.g., `sm1` and `sm2` in Figure 7.6). A *deep* comparison is performed at the bottom of the tank. A deep comparison compares the contents of two objects. The methods that perform shallow and deep comparisons are fundamentally different and will be discussed separately.

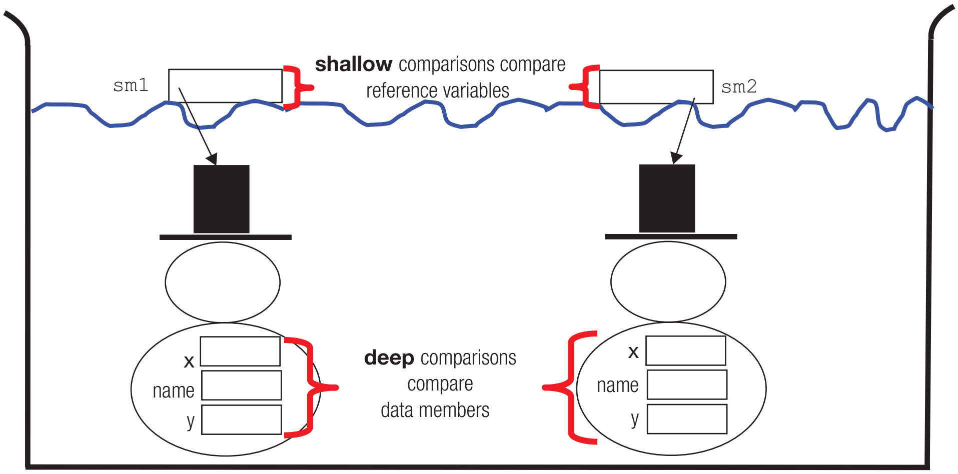


Figure 7.6
A fish tank analogy of shallow and deep comparisons.

7.3.1 Shallow Comparisons

In Chapter 4, we used the relational operators to compare two primitive values. For example, to determine if the values stored in the integer variables `age1` and `age2` are equal, we used the equality operator (==):

```
if(age1 == age2)
```

This comparison is a *shallow comparison* because primitive variables contain one value, so they would float on the top of a fish tank. Because reference variables also float, the same syntax used to perform a shallow comparison of two primitive values can be used to perform a shallow comparison of two objects. In effect, a shallow comparison of two objects determines if two reference variables refer to the same object.

Normally, a method named `equals`, such as the method `equals` in the `String` class, performs a *deep* comparison of two objects. It compares information contained inside the objects. For this reason, when coding a shallow `equals` method in the class of the objects being compared it is good coding practice to name the method `shallowEquals`. This name clearly indicates that the method is making a shallow comparison. The following method, which would be coded inside the class `ParentSnowman`, performs a shallow comparison of the `ParentSnowman` object that invoked it and the object passed to its parameter. It uses the equality relational operator to perform the comparison.

```
public boolean shallowEquals(ParentSnowman ps) //a shallow comparison
{
  if(this == ps) //this contains the address of the invoking object
```

```
    {
     return true; //the invoking object and ps refer to the same object
    }
    else
    {
      return false; //ps does not reference the invoking object
    }
  }
```

An exception to this comparison-method naming convention is the method `equals` in the API class `Object`. It performs a shallow comparison of two objects. The following code fragment illustrates the use of this shallow comparison method. It outputs the Boolean value `true` and then `false` because the variables `ps1` and `ps2` contain the same address and `ps1` and `ps3` do not.

```
ParentSnowman ps1 = new ParentSnowman();
ParentSnowman ps2 = ps1; //ps2 is initialized to the address in ps1
ParentSnowman ps3 = new ParentSnowman();
boolean sameAddresses;

sameAddresses = ps1.equals(ps2); //shallow comparison, returns true
System.out.println(sameAddresses);
sameAddresses = ps1.equals(ps3); //shallow comparison, returns false
System.out.println(sameAddresses);
```

7.3.2 Deep Comparisons

Deep comparisons compare the contents of two objects' data members. As previously mentioned, before we write a method that performs a deep comparison for a particular application, we have to determine which of the objects' data members to compare for that application. For example, if we decide that two snowmen are equal if they are at the same game board position, then the x and y data member of the two objects would be compared. Once this decision is made, the class's `get` methods are used to fetch the data members, and they are compared using the relational operators if they are primitive variables. If they are reference variables that refer to other objects, they are compared using the deep comparison method in the objects' class.

The method shown in Figure 7.7, which would be coded in the class `ParentSnowman`, performs a deep comparison of two `ParentSnowman` objects: the object that invokes it and the object passed to its parameter. As depicted in Figure 7.6, each `ParentSnowman` object contains its (x, y) location and a reference to its family name (a string). The method returns `true` when the two objects are at the same game board (x, y) location *and* the objects' family name, referenced by string data member `name`, are the same.

```
public boolean equals(ParentSnowman ps) //a deep comparison
{
  if(x == ps.getX() && y == ps.getY() &&
     name.equals(ps.getName()))
  {
```

```
6       return true; //same location and family name
7     }
8     else
9     {
10      return false; //different location and/or family name
11    }
12 }
```

Figure 7.7
A deep comparison method named **`equals`** that would be part of the **`ParentSnowman`** class.

It should be noted that the invocation of the `equals` method on line 4 of Figure 7.7 is an invocation of the `String` class's `equals` method. This method is invoked because `name` is a string variable and the argument that follows its name on line 4, `ps.getName()`, returns a reference to a `String` object.

There are four common errors made when coding the third term of the Boolean condition on lines 3 and 4 of Figure 7.7:

```
1. name == ps.getName()
2. this.getName() == ps.getName()
3. name.equals(ps)
4. this.equals(ps)
```

When either of the first two errors is made, a shallow comparison of the `name` data members is performed, and the Boolean condition evaluates to `false` even when the strings are the same. The two addresses, not the strings, are being compared. When the third error is made, a string is being compared to a `ParentSnowman` object, and the `Object` class's `equals` method is invoked, which also makes a shallow comparison.

When the fourth error is made, the program ends in a runtime *StackOverflow* error. Two `ParentSnowman` objects are being compared causing line 4 to repeatedly invoke the same method of which it is a part. This is a concept called recursion. It is not considered a syntax error because when recursion is properly used, it can facilitate the coding of many algorithms. We will learn more about the proper use of this coding technique in Chapter 9.

Deep comparisons are also performed to compare the relative order of two objects. A method that compares two objects to determine their relative order is normally named `compareTo`, and it returns an integer. The string class contains a method named `compareTo` that determines the lexicographical order of two `String` objects. Aside from changing the name of the method presented in Figure 7.7 and the returned type and value, `compareTo` methods would use the less than (<) or the greater than (>) operator to compare primitive data members and use previously coded `compareTo` methods to compare data members that are objects.

7.4 COPYING AND CLONING OBJECTS

The deep and shallow fish tank analogy also helps us to understand the techniques used to copy and clone objects. We can make both shallow and deep copies of objects, and clones of objects can easily be created from deep copies of objects. Figure 7.8 illustrates the difference between a shallow and a deep copy. Shallow copies copy the contents of one object's reference variable into another object's reference variable. Deep copies copy all of the values of one object's data members into the data members of another object. We will begin with a discussion of shallow copies.

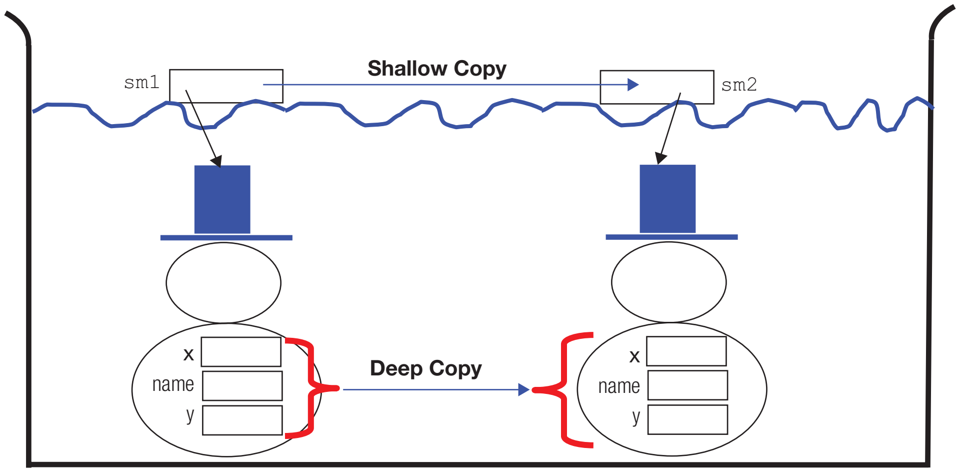


Figure 7.8
A fish tank analogy of shallow and deep copies.

7.4.1 Shallow Copies

In Chapter 2, we used the assignment operator to copy the contents of one primitive variable into another. For example, the following line of code copies the value stored in the integer variable `age1` into the variable `age2`:

```
age2 = age1;
```

This statement actually performs a shallow copy between two primitive variables because primitive variables contain only one value, so they would float on the top of a fish tank. Because reference variables also float (contain one value, an address) and the assignment operator can be used to assign reference variables, the same syntax is used to make a shallow copy of an object. The following line of code makes a shallow copy of object `sm1` into object `sm2`:

```
sm2 = sm1; //a shallow copy of object sm1 into object sm2
```

Although we say we are making a shallow copy of an object, we are actually copying the contents of one object's reference variable into the other object's reference variable, as shown at the top of Figure 7.8. It is important that we understand the consequences of this.

Referring to Figure 7.8, after the shallow copy of object `sm1` into `sm2` is complete, the address of the snowman object on the right side of the fish tank that was stored in the reference variable `sm2` has been overwritten with the address of the snowman object on the left side of the fish tank. Now both reference variables, `sm1` and `sm2`, refer to the *same* object. As a result, the snowman on the left now has two names. In addition, the snowman on the right side of Figure 7.8 is no longer part of the program (unless another variable in the program also stored its address) because the address of the object is no longer known to the program. When an object's address is not stored in a program reference variable, the Java memory manager reclaims its storage.

NOTE *A shallow copy gives one object two names, and it may eliminate an object from the program.*

The following code fragment makes a shallow copy of object `ps1` into `ps2`, and as a result outputs the data members of the object declared on line 1 twice:

```
1  ParentSnowman ps1 = new ParentSnowman(250, 250, "A");
2  ParentSnowman ps2 = new ParentSnowman(10, 20, "X");
3
4  ps2 = ps1; // makes a shallow copy of object ps1 into object ps2
5  System.out.println(ps1.show());
6  System.out.println(ps2.show());
```

7.4.2 Deep Copies and Clones

As shown in the bottom of Figure 7.8, when we make a deep copy of an object, the values of its data members are copied into the data members of another object. Unlike performing a shallow copy, when a deep copy is complete both objects *still exist*, and the values stored in their data members are identical. The method `arraycopy` discussed in Section 6.9.1 can be used to deep copy one array object into another.

Generally speaking, a method has to be added to a class to be able to make deep copies of instances of the class. A deep copy method uses the class's `set` methods to copy the data members of the object that invoked it into the data members of the object sent to its parameter.

The method shown in Figure 7.9 is a deep copy method, which would be coded in the class `ParentSnowman`. As depicted in Figure 7.8, each `ParentSnowman` object contains its (x, y) location and the object's family name, referenced by the string data member `name`:

```
public void deepCopy(ParentSnowman ps) //a deep copy into ps method
{
  ps.setX(x);
  ps.setY(y);
  ps.setName(name);
}
```

Figure 7.9
A **`deepCopy`** method that would be part of the **`ParentSnowman`** class.

In some cases, it is desirable to deep copy a subset of one object's data members into another object. When this is the case, it is good programming practice to name the method `copy`, rather than `deepCopy`, to alert the reader of the method's signature to the fact that not all of an object's data members are being copied.

Cloning Objects

When an object is cloned, a new instance of the object's class is *created*, and the values of all of an existing object's data members are copied into the corresponding data members of the new object. If one object existed before the clone was created, *two* objects exist after it is created. To create a clone of an object, a method (usually named `clone`) is coded in the object's class. The method is a nonvoid method that returns the address of the newly created clone object.

Figures 7.10 and 7.11 show two alternate codings of a clone method, which would be coded in the class `ParentSnowman`. Both methods return the address of a newly created clone of the object that invoked them. The version shown in Figure 7.10 invokes the class's `deepCopy` method on line 4 to copy the values of the data members of the object that invoked the method into the clone object created on line 3. This version of the method assumes that the class contains a deep copy method and a no-parameter constructor. The new object is identical to (an exact copy of) an existing object.

```
1  public ParentSnowman clone() //a deep copy
2  {
3    ParentSnowman theClone = new ParentSnowman();
4    this.deepCopy(theClone);
5    return theClone;
6  }
```

Figure 7.10
A clone method using a **`deepCopy`** method that would be part of the class **`ParentSnowman`**.

The alternate coding of the clone method presented in Figure 7.11 uses the class's three-parameter constructor on line 3 to copy the values of the data members into the clone when it is constructed. If each object only contains three data members, then the newly created object is identical to (an exact copy of) an existing object. Generally speaking, to produce an exact copy of an object with `n` data members, this version of the clone method would have to invoke an `n`-parameter constructor on line 3 of Figure 7.11.

```
1  public ParentSnowman clone() //a deep copy
2  {
3    ParentSnowman theClone = new ParentSnowman(x, y, name);
4    return theClone;
5  }
```

Figure 7.11
An alternate clone method that would be part of the class **`ParentSnowman`**.

If it is appropriate to a particular application for the clone method to copy only a subset of an object's data members into the newly created object, then line 4 of Figure 7.10 would be changed to an invocation of the class's `copy` method, and the clone method would be renamed `partialClone` to indicate that invocations of this method do not make an identical copy of the object that invoked it.

Figure 7.12 is the code of a class named `ParentSnowmanV2` that is an expanded version of the `ParentSnowman` class shown in Figure 6.18. The expanded version adds the following methods to the class:

- a `copy` method (lines 29–35) that copies four of an object's data members into another object
- a partial-clone method (lines 36–41) that invokes the `copy` method
- a shallow compare method (lines 42–52)
- a deep compare method (lines 53–63) that compares the `hatColor` data members of two objects
- a method that detects collisions (lines 64–75) between two snowmen

In addition, three static data members (lines 5–7) and a `get` method (lines 93–96) to fetch one of these data members have also been added to the class.

As discussed in Section 7.1, the three static data members declared on lines 5, 6, and 7 of Figure 7.12 are shared by all instances of this class. The first of these, `snowmanCount` initialized to zero, will be used to count the number of snowmen constructed. It is incremented inside the no-parameter constructor (line 18) and the four-parameter constructor (line 27) every time these constructors execute. The client can fetch the value of the variable `snowmanCount` using the `get` method coded on lines 93–96.

The other two static variables, `w` and `h`, store the width (40) and height (77) of a snowman, as depicted in Figure 4.9. Because the class's `show` method draws all snowmen with the same width and height, it is appropriate that these two variables be shared by all instances of the class. The variables are used in the Boolean condition coded within the `collidedWith` method (lines 64–75) that detects a collision between the snowman that invoked it and the snowman passed to its parameter `ps`. The values of the variables `x` and `y` used on lines 66 and 67, and within other methods of the class, are the values of the `x` and `y` data members of the snowman that invoked the method. Alternately, we could code these data members as `this.x` or `this.y`.

The class's `copy` method (lines 29–35) copies four of a snowman's seven nonstatic data members (lines 31–34). When the method completes its execution, the snowman passed to its parameter will have the same name, hat color, and location as the snowman that invoked the method. To prevent methods that are not part of the `ParentSnowmanV2` class from erroneously invoking this method to make a deep copy of *all* of an object's data members, it is declared as a private method on line 29.

The `copy` method is invoked on line 39 by the class's `partialClone` method, which means that clones will have only the same name, hat color, and location as the snowman from which they were cloned. Their `xSpeed`, `ySpeed`, and `visible` data members will retain their default values (set on lines 12–14). As previously discussed in Section 7.2, the fact that the `copy` method is a private method does not prevent the class's `clone` method from invoking it.

The deep comparison performed on line 55 of the `equals` method determines if two snowmen have the same hat color by invoking the API `Color` class's `equals` method. If they do, the method returns the value `true`.

```
import java.awt.*;

public class ParentSnowmanV2
{
  private static int snowmanCount = 0;
  private static int w = 40;
  private static int h = 77;
  private int x = 8;
  private int y = 30;
  private String name;
  private Color hatColor= Color.BLACK;
  private int xSpeed = 2;
  private int ySpeed = 2;
  private boolean visible = true;

  public ParentSnowmanV2()
  {
    snowmanCount++;
  }
  public ParentSnowmanV2(int intialX, int intialY, String name,
                         Color hatColor)
  {
    x = intialX;
    y = intialY;
    this.name = name;
    this.hatColor = hatColor;
    snowmanCount++;
  }
  private void copy(ParentSnowmanV2 ps) //copies 4 data members
  {
    ps.setX(x);
    ps.setY(y);
    ps.setName(name);
    ps.setHatColor(hatColor);
  }
  public ParentSnowmanV2 partialClone()
  {
    ParentSnowmanV2 theClone = new ParentSnowmanV2();
    this.copy(theClone);
    return theClone;
  }
  public boolean shallowEquals(ParentSnowmanV2 ps)
  {
    if(this == ps)
    {
      return true;
    }
    else
    {
```

```
        return false;
      }
    }
    public boolean equals(ParentSnowmanV2 ps)
    {
      if(hatColor.equals(ps.getHatColor())) //same hat color
      {
        return true;
      }
      else
      {
        return false;
      }
    }
    public boolean collidedWith(ParentSnowmanV2 ps)
    {
      if( !(x > ps.getX( ) + w || x + w < ps.getX( ) ||
            y > ps.getY( ) + h || y + h < ps.getY( )))
      {
       return true;
      }
      else
      {
       return false;
      }
    }
    public void show(Graphics g) // g is the game board object
    {
      int[] xPoly = {x + 20, x + 15, x + 25};
      int[] yPoly = {y + 25, y + 30, y + 30};

      g.setColor(hatColor);
      g.fillRect(x + 15, y, 10, 15);  // hat
      g.fillRect(x + 10, y + 15, 20, 2);  // brim
      g.setColor(Color.WHITE);
      g.fillOval(x + 10, y + 17, 20, 20); // head
      g.fillOval(x, y + 37, 40, 40);  // body
      g.setColor(Color.RED);
      g.fillPolygon(xPoly, yPoly, 3); // nose
      g.setColor(Color.BLACK);
      g.setFont(new Font("Arial", Font.BOLD, 16));
      g.drawString(name, x + 16, y + 62); // name
    }
    public static int getSnowmanCount()
    {
      return snowmanCount;
    }
    public int getXSpeed()
    {
```

```
    return xSpeed;
  }
  public void setXSpeed(int newXSpeed)
  {
    xSpeed = newXSpeed;
  }
  public int getYSpeed()
  {
    return ySpeed;
  }
  public void setYSpeed(int newYSpeed)
  {
    ySpeed = newYSpeed;
  }
  public void setHatColor(Color newHatColor)
  {
    hatColor = newHatColor;
  }
  public Color getHatColor()
  {
    return hatColor;
  }
  public int getX()
  {
    return x;
  }
  public void setX(int newX)
  {
    x = newX;
  }
  public int getY()
  {
    return y;
  }
  public void setY(int newY)
  {
    y = newY;
  }
  public String getName()
  {
    return name;
  }
  public void setName(String newName)
  {
    name = newName;
  }
  public boolean getVisible()
  {
```

```
    return visible;
  }
}
```

Figure 7.12
The class **ParentSnowmanV2**.

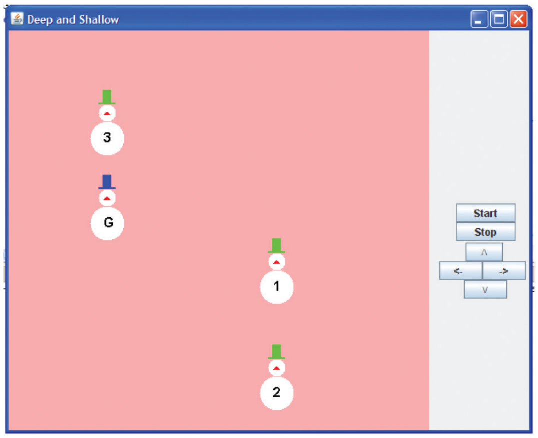
The application shown in Figure 7.13 uses most of the concepts discussed up to this point in this chapter. When the game board's Start button is clicked, a snowman guard, whose family name is "G" patrols his game board garden looking for three green-hat snowmen who have wandered into the garden (Figure 7.14a). When he finds (collides with) one, he positions the green-hat snowman behind himself and continues his search (Figure 7.14b). Each time he reaches the border of his garden, he clones himself and posts the clone at the garden's edge (Figure 7.14c) to guard the garden from wandering snowmen. After the guard has found the three green-hat snowmen and posted six clones at the garden's boundaries, the animation ends (Figure 7.14d).

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class DeepAndShallow extends DrawableAdapter
{ static DeepAndShallow ge = new DeepAndShallow ();
  static GameBoard gb = new GameBoard(ge, "Deep and Shallow");
  static ParentSnowmanV2[] ps;
  static boolean gameOver = false;

  public static void main(String[] args)
  { ps = new ParentSnowmanV2[10];
    ps[0] = new ParentSnowmanV2(100, 200, "G", Color.BLUE);
    ps[1] = new ParentSnowmanV2(300, 275, "1", Color.GREEN);
    ps[2] = new ParentSnowmanV2(300, 400, "2", Color.GREEN);
    ps[3] = new ParentSnowmanV2(100, 100, "3", Color.GREEN);

    gb.setTimerInterval(3, 20);
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 1; i < ps.length; i++)
    {
      if(ps[i] != null) //the snowman exists
      {
        ps[i].show(g);
      }
    }
    ps[0].show(g); //the patrolling guard
  }

  public void timer3()
```

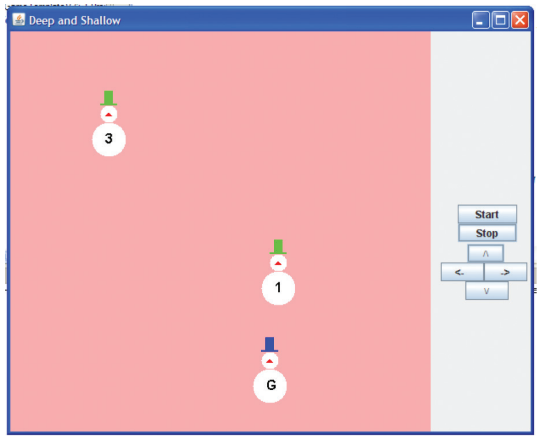
```
    { int x, speed, y;
      if(ParentSnowmanV2.getSnowmanCount()== 10)
      {
        gb.stopTimer(3);
        gameOver = true;
      }
      //move the guard
      x = ps[0].getX();
      x = x + ps[0].getXSpeed();
      ps[0].setX(x);
      y = ps[0].getY();
      y = y + ps[0].getYSpeed();
      ps[0].setY(y);

      //is ps[0] at a border?
      if(ps[0].getX() >= 460 || ps[0].getX() <= 6)
      {
        speed = ps[0].getXSpeed();
        speed = -speed;
        ps[0].setXSpeed(speed);
        ps[ParentSnowmanV2.getSnowmanCount()] = ps[0].partialClone();
      }
      if(ps[0].getY() >= 423 || ps[0].getY() <= 30)
      {
        speed = ps[0].getYSpeed();
        speed = -speed;
        ps[0].setYSpeed(speed);
        ps[ParentSnowmanV2.getSnowmanCount()] = ps[0].partialClone();
      }

      // has ps[0] found a green-hat wandering snowman?
      for(int i = 1; i <= ps.length; i++)
      {
        if(ps[i] != null && ps[0].collidedWith(ps[i]) &&
           !ps[0].equals(ps[i]))
        {
          ps[i].setX(ps[0].getX()); //position wanderer behind ps[0]
          ps[i].setY(ps[0].getY());
        }
      }
    }

    public void leftButton()
    {
      if(gameOver == true)
      { for(int i=0; i<=3; i++) //move the three intruders left
        {
          ps[i].setX(ps[i].getX() - (i * 3));
```

```
83              }
84              ps[0].setX(ps[0].getX() - 1); // move the guard
85          }
86      }
87  }
```

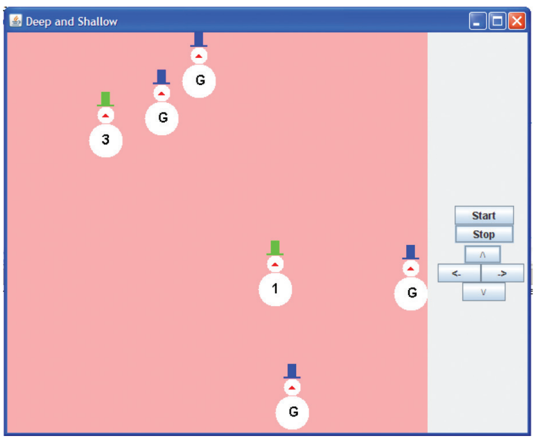
Figure 7.13
The application **`DeepAndShallow`**.



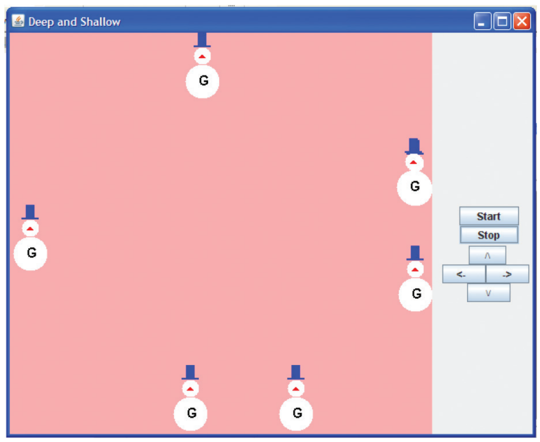
(a) Game board when the program is launched



(b) Game board after green-hat snowman 2 is found and placed behind the guard



(c) Game board after three clones are posted and green-hat snowman 3 is about to be found



(d) Game board after six clones have been posted and all the wonderers have been found

Figure 7.14
Graphical output produced by the application **`DeepAndShallow`**.

This animation uses an array of ten `ParentSnowmanV2` snowmen objects, whose class is shown in Figure 7.12; the array, `ps`, is declared on line 11 of Figure 7.13. The patrolling snowman guard and the three green-hat snowmen, declared on lines 12–15, are referenced by indices 0 and 1–3, respectively. The remaining six elements of the array will be used to reference the clone snowmen.

The patrolling of the guard and the game termination are dependent on `timer3`. Line 17 of the main method sets `timer3`'s increment to 20 milliseconds, which means that the `timer3` call back method (lines 34–75) executes every 20 milliseconds. Its code determines if the patrol is over (lines 36–44), moves the patrolling guard (lines 42–57), posts clone guards (lines 50–63) and gathers up the intruders (lines 66–74). Line 36 determines if ten snowmen have been constructed, which would mean that all six guards have been posted. If so, the patrol is ended by line 38, which stops `timer3`. While the timer is ticking, lines 42–47 use the `Snowman` class's `set` and `get` methods to fetch, change, and overwrite the current values of the patrolling guard's (x, y) location every 20 milliseconds. This keeps the snowman on patrol.

Lines 50 and 57 determine if the guard has reached a vertical (line 50) or horizontal (line 57) edge of the garden. If it has, lines 52–54 and lines 59–61 reflect the guard off the edge by changing the sign of the x and/or y speed data member. Then, lines 55 and 62 invoke the `partialClone` method to create the clone snowman and store its returned address in the next available element of the array `ps`. The index of this element is the static value returned from the `getSnowmanCount` method, which is invoked on the left side of these lines. Because the `partialClone` method in the `ParentSnowmanV2` class (Figure 7.12) copies the current (x, y) position of the patrolling snowman into the clone, the clone is positioned at an edge of the garden.

Just before the `timer3` method ends, the `for` loop that begins on line 66 indexes through the last nine elements of the array. For each of these elements, the `if` statement's Boolean condition on lines 68 and 69 determines if:

- the element of the array references a snowman (is not `null`)
- the referenced snowman has collided with the patrolling guard (`ps[0]`) as determined by the `collidedWith` method
- the referenced snowman is an intruder as determined by the `equals` method (Its hat color is not equal to the guard's hat color.)

When this is the case, the (x, y) position of the green-hat wandering snowman is set to the (x, y) position of the patrolling guard (lines 71–72). On the next tick of `timer3`, the patrolling guard is only moved two pixels, so the guard and the snowman remain in a collided state, and the wandering snowman's position is reset by lines 74–75 to the patrolling guard's new position. The resulting effect is that the wandering snowman goes on patrol with the guard. It cannot be seen on patrol because line 31 of the `draw` method draws the patrolling guard, `ps[0]`, last (i.e., on top of the green-hat snowmen that have joined the patrol).

After the game ends, a series of left button clicks will separate the patrolling guard from the green-hat guards that have joined the patrol as shown in the center portion of Figure 7.15. This is accomplished by decrementing the x coordinates of each green-hat patrolling snowman (line 82) and the guard (line 84) by a different amount on each button click. Until the game ends, the left button is inactive because the body of the `if` statement on line 79 that separates the snowmen is only executed when the Boolean variable `gameOver` is set to `true` on line 39 of the `timer3` call back method.

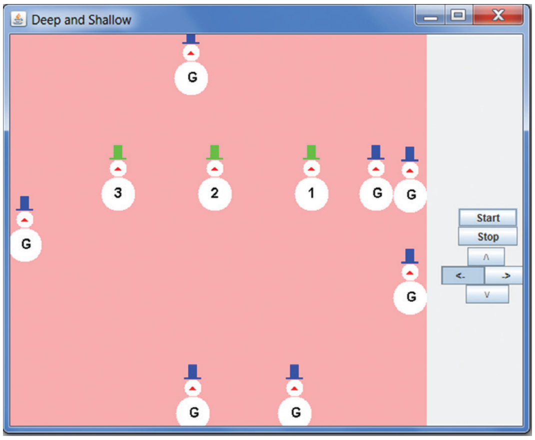


Figure 7.15
Graphical output produced by the application **DeepAndShallow** after the animation ends and several left button clicks have been performed.

7.5 THE STRING CLASS: A SECOND LOOK

In Section 2.5, we discussed the creation of `String` objects, and in Section 4.2.3, we discussed the `String` class's methods `equals` and `compareTo`, which are used to compare two strings. In addition to these methods, the class contains a rich collection of methods used to create strings, to convert strings to characters, and to process string data.

7.5.1 Creating Strings from Primitive Values

Aside from the one-parameter constructor discussed in Chapter 2 that is passed a string literal or a string object such as

```
String name1 = new String("Robert");
String name2 = new String(name1);
```

the `String` class also contains several overloaded constructors. Each of these creates a new string object and returns its address. Two of the most frequently used constructors have a character array passed to them. The following code fragment uses the one-parameter version of these two methods to create the string `name1` initialized to *Joanne Jones* and the three-parameter version to create the string `name2` initialized to *anne Jo*. (The second argument sent to the three-parameter version of the constructor would be *zero* to include the first character of the array in the string object.)

```
char[] c = {'J', 'o', 'a', 'n', 'n', 'e', ' ', 'J', 'o', 'n', 'e', 's'};
String name1 = new String(c); //all characters used
String name2 = new String(c, 2, 7); //7 characters used, starts @ index 2
```

The static method `valueOf` in the `String` class also can be used to create and return the address of a string object initialized to the argument passed to it. Overloaded versions of this method use a `char`, `int`, `long`, `float`, or `double` parameter. Integer type `byte` and `short` arguments can be passed to the `int` parameter version of this method. In addition, there is a version of the method that accepts a character array passed to it. The following code fragment demonstrates the use of this overloaded method, to output the string: *x102030405.556.66yz*.

```
char c = 'x';
byte b = 10;
short s = 20;
int i = 30;
long l = 40;
float f = 5.55f;
double d = 6.66;
char[] cArray = {'y', 'z'}

String s2 = String.valueOf(c) + String.valueOf(b) + String.valueOf(s) +
            String.valueOf(i) + String.valueOf(l) + String.valueOf(f) +
            String.valueOf(d) + String.valueOf(cArray);

System.out.println(s2);
```

7.5.2 Converting Strings to Characters

The `String` class method `charAt` can be used to convert a particular character contained in the string object that invoked it to type `char`, and it returns the character. The following code fragment stores the character `'S'` in the variable `c`.

```
String aString = "Joe Smith";
char c = aString.charAt(4);  //character numbers begin at zero
```

The method `toCharArray` converts a string object that invokes it to an array of characters and returns the address of the array. After the following code fragment executes, the character arrays `c1` and `c2` contain the same character sequence:

```
char[] c1 = {'J', 'o', 'a', 'n', 'n', 'e', ' ', 'R', 'a', 'y'};
char[] c2;
String aString = "Joanne Ray";
c2 = aString.toCharArray();
```

7.5.3 Processing Strings

The `String` class contains several methods to process string data. Two of these methods, `compareTo` and `equals` (and their related methods `compareToIgnoreCase` and `equalsIgnoreCase`) were discussed in Section 4.2.3. Table 7.1 describes other `String` class methods often used to process strings.

All of these methods are nonstatic methods. They can be used to determine the existence or location of a given substring in a string, fetch a substring from a specified position in the string, produce an uppercase or lowercase version of a string, replace the first or all occurrences of a substring with a given string, and tokenize a string to determine if a string begins or ends with a specified string.

Table 7.1
String Processing Methods in the String Class

Function	Method and Parameter Type	Example and Description
Locate a given substring in a string	`indexOf` Parameter(s): String *or* String, `int`	`int i = s1.indexOf(target);` Returns the index of the *first* occurrence of substring `target` in the string `s1`
		`int i = s1.indexof(target, start);` Returns the index of the first occurrence of substring `target` that begins at or after the index `start` in the string `s1`
Fetch a substring from a string	`substring` Parameter(s): `int` *or* `int`, `int`	`String s2 = s1.substring(start);` Returns the substring of `s1` from index `start` to the end of `s1`
		`String s2 = substring(start, end);` Returns the substring of `s1` from index `start` to index `end -1`
Convert a string to upper or lower case characters	`toUpperCase`	`String s2 = s1.toUpperCase();` Creates a clone of string `s1` consisting of all uppercase characters
	`toLowerCase`	`String s2 = s1.toLowerCase()` Creates a clone of string `s1` consisting of all lowercase characters
Replace a given substring in a string with a given substring	`replaceFirst` Parameter(s): String, String	`String s2 = s1.replaceFirst(target, new);` Returns a string with the first occurrence of the substring `target` in `s1` replaced with the string `new`
	`replaceAll` Parameter(s): String, String	`String s2 = s1.replaceAll(target, new);` Returns a string with all occurrences of the substring `target` in `s1` replaced with the string `new`
Determine if a string begins or ends with a given string	`startsWith` Parameter: String	`boolean b = s1.startsWith(s2);` Returns *true* if the string `s1` begins with the string `s2`
	`endsWith` Parameter: String	`boolean b = s1.endsWith(s2);` Returns *true* if the string `s1` ends with the string `s2`
Tokenize a string	`split` Parameter(s): String or String, `int`	`String[] s1 = s1.split(" +");` Returns a string array whose elements are the substrings of `s1` that are separated by one or more spaces
		`String[] s1 = s1.split(" +" , n);` Returns a string array whose elements are the substrings of `s1` that are separated by one or more spaces; there will be `n` or fewer elements in the returned array

The application `StringProcessing` presented in Figure 7.16 uses some of the methods described in Table 7.1 to process a sentence input by the user. If the word *Hello* appears at the beginning of the sentence it is eliminated, and if the word *Tom* is in the sentence it is changed to *XXX*. A typical input and corresponding output are given in the left and right sides of Figure 7.17, respectively.

Line 16 uses the `String` class method `split` to create a new `String` array and place each word of the sentence input on line 11 into its elements. Then, it stores the address of the array returned from `split` in the variable `sArray` that is declared on line 8. The string consisting of a space followed by a plus sign passed to `split` on line 16, instructs the method to consider one or more spaces as token delimiters in the sentence. Line 19 clones the string input on line 11 and stores its address in the variable `s2`.

Lines 20–23 remove the word *Hello* from the beginning the cloned string `s2`. The `indexOf` method is used in the `if` statement's Boolean condition on line 20 to determine if the substring "*Hello*" begins at the index zero of the string `s2`. If it does, the `substring` method is used on line 22 to return the substring of `s2` that begins at index 6 (after the word *Hello* and the space that follows it). The address of this new string object is stored in (shallow copied into) `s2`. Line 20 could have been coded as shown below:

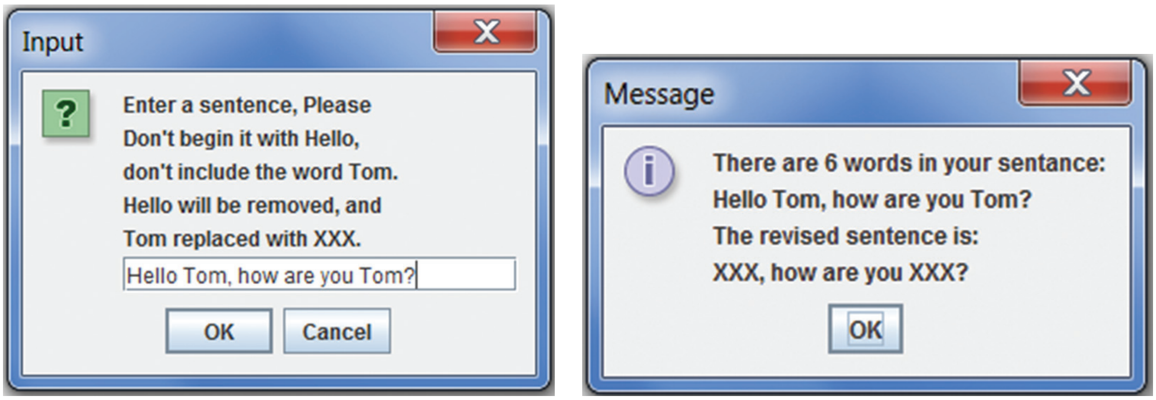
```
if(s2.startsWith("Hello"))
```

The `replaceAll` method is invoked on line 24. It effectively creates a clone of the string object `s2` and replaces all occurrences of the word *Tom* with the string literal "XXX." The returned address of the modified clone is then assigned to (shallow copied into) the variable `s2`.

```
1    import javax.swing.*;
2
3    public class StringProcessing
4    {
5      public static void main(String[] args)
6      {
7        String s1, s2;
8        String[] sArray;
9        int nWords;
10
11       s1 = JOptionPane.showInputDialog("Enter a sentence, Please\n" +
12                                        "Don't begin it with Hello,\n" +
13                                        "don't include the word Tom.\n" +
14                                        "Hello will be removed, and \n" +
15                                        "Tom replaced with XXX.");
16       sArray = s1.split(" +"); //stores each word in separate elements
17       nWords = sArray.length; //the number of words
18
19       s2 = new String(s1);
20       if(s2.indexOf("Hello") == 0)
21       {
```

```
      s2 = s2.substring(6);
    }
    s2 = s2.replaceAll("Tom", "XXX");

    JOptionPane.showMessageDialog(null, "There are " + nWords +
                                  " words in your sentence:\n" +
                                  s1 + "\nThe revised sentence is:\n" +
                                  s2);
  }
}
```

Figure 7.16
The application **`StringProcessing`**.



(a) Input (b) Output

Figure 7.17
Input to the application **`StringProcessing`** and the corresponding output.

Tokenizing a String

Generally speaking groupings of sequential characters within a string, which are most often separated by white-space (e.g., a tab, a new line, or space keystrokes), are called tokens. Tokenizing a string is the process of separating a string into its component tokens. The application shown in Figure 7.16 use the `String` class's `split` method to perform this process.

Another way of tokenizing a string is to declare an instance of the `StringTokenizer` class and pass its constructor the string to be tokenized. The class's `nextToken` method is repeatedly invoked on the `StringTokenizer` object within a while loop whose code block processes the tokens. This method sequentially returns one token per invocation, beginning with the token on the left side of the string. The class's `hasMoreTokens` method returns `false` if all of the tokens in its `StringTokenizer` object have been fetched.

The following code outputs *token1 token2 token3* regardless of the width, or type, of the white-space between the tokens.

```
String s = "token1   token2          token3";
StringTokenizer tokens = new StringTokenizer(s);
```

```
while(tokens.hasMoreTokens())
{
   System.out.print(tokens.nextToken() + " " );
}
```

7.6 THE WRAPPER CLASSES: A SECOND LOOK

As discussed in Section 2.7.3, the six numeric wrapper classes (e.g., `Integer`, `Double`, etc.) contain methods that parse a string into integer or real values that can then be stored in primitive variables. In addition to these methods, all of the wrapper classes contain constructors to create a wrapper class object and also contain a collection of useful constants. A seventh wrapper class, the class `Character`, contains methods to perform common operations on characters. In this section, we will discuss these additional features of the wrapper classes and a related topic called autoboxing.

7.6.1 Wrapper Class Objects

A wrapper class's one-parameter constructor can be used to create a wrapper object. Wrapper class objects contain a single private primitive data member whose type is consistent with the class's name. For example, an object in the class `Double` contains a `double` primitive data member, and an instance of an `Integer` contains an `int` data member. The argument passed to the one-parameter constructor is stored in the object's data member. Envisioning the primitive value being wrapped in the object is the basis of the phrase *wrapper classes*.

The wrapper classes contain methods that return the value stored in an object's data member. These methods perform the function of `get` methods, but their names begin with the primitive type they return followed by the word *Value*. For example, the `Integer` class contains a method named `intValue` that returns the value of the class's integer data member, and the `Character` class contains the method `charValue` that returns the value of the class's character data member.

The `Integer` class also contains the methods `byteValue`, `shortValue`, `longValue`, `floatValue`, and `doubleValue` that cast the returned value of the data member into the other numeric primitive types. The other numeric wrapper classes contain five similarly named methods used to cast the data member they return into different numeric types. The following code segment wraps the value 20 in an `Integer` object and outputs the value as an integer and a double (e.g., 20 followed by 20.0):

```
Integer n = new Integer(20);
System.out.println(n.intValue() + " " +
                   n.doubleValue()); //outputs: 20 20.0
```

Each of the wrapper classes contains an `equals` and a `compareTo` method to perform a deep comparison of the data wrapped in the object that invoked them and the object sent to their parameter. The `equals` method returns the Boolean value `true` when the two objects are equal, otherwise it returns `false`. The equality operator (==) can be used to perform a shallow comparison to determine if two wrapper class reference variables refer to the same object.

The compareTo method returns an integer, which is negative, zero, or positive when the invoking object is less than, equal to, or greater than the object sent to its parameter, respectively. This version of the method coded in the numeric wrapper classes always returns -1, 0, or 1. The integer returned from the Character class's version of the method also gives an indication of the lexicographic separation of the characters contained in the objects being compared.

The following code fragment outputs the value -25. The value is negative because the character 'a' appears before 'z' in the Unicode table, and its absolute value is 25 because 'a' is 25 characters before 'z'.

```
Character c1 = new Character('a');
Character c2 = new Character('z');
System.out.println(c1.compareTo(c2));  //outputs: -25
```

The wrapper classes do not contain set methods to change the value of the data member wrapped in the object. This is because wrapper class objects, like String objects, are *immutable*. Once a value has been stored in a wrapper class's data member or inside a String object, the value cannot be changed. As is the case with String objects, the assignment operator can be used to reassign the address stored in the object's reference variable to a newly created object that contains the assigned (different) value. Although this gives the appearance that the value stored in the object has changed, in reality, the new value has been stored in a different object.

The assignment operator can also be used to shallow copy (the address of) two wrapper class objects. The following code fragment outputs the value 12.5 twice. Although it initially creates two instances of a Double, d1 and d2, after the third line executes, these variables reference the same object. The object containing the value *54.6* is reclaimed by the Java memory manager. The coding of the variables d1 and d2 in the argument sent to the println method are two implicit invocations of the Double class's toString method.

```
Double d1 = new Double(12.5);
Double d2 = new Double(54.6);
d2 = d1; //d1 and d2 reference the same object which contains 12.5;
System.out.println(d1 + " " +  d2); //outputs: 12.5 12.5
```

7.6.2 Autoboxing and Unboxing

The autoboxing feature of wrapper classes makes it easier to use wrapper class objects in our programs. This feature automatically "wraps" primitive values into wrapper objects. For example, wrapper class objects can be declared using the abbreviated syntax used to declare String objects, as discussed in Section 2.5. The following line of code wraps, or boxes, the integer 20 in an Integer object and writes its address in the variable n1:

```
Integer n1 = 20; //autoboxing of the value 20 in an object declaration
```

When this statement executes, the autoboxing feature creates an Integer wrapper object, stores (boxes) 20 in its data member, and returns the object's address. The statement is equivalent to the following statement:

```
Integer n1 = new Integer(20);
```

Autoboxing can also be used to effectively reassign or set the primitive value stored in a wrapper class object. The following code fragment outputs the value 3.6. The right side of line 2 uses autoboxing to create a new `Integer` wrapper object, stores 3.6 in its data member, and returns its address. The returned address is then stored in the variable `n1`. The wrapper object containing the value 2.5 is reclaimed by the Java memory manager.

```
1     Double n1 = 2.5;
2     n1 = 3.6; //Autoboxing of the value 3.6 in an assignment statement
3     System.out.println(n1);
```

It should be noted that if the value to be boxed in a wrapper class object (e.g., `n2`) is already stored in another wrapper class object (e.g., `n1`) of the same type, then a new object is not created. Rather, `n2` is set to refer to `n1's` object. The only exception to this is if the long form of the object-declaration grammar is used to declare and initialize the new object. This caveat also applies to `String` objects. The code fragment shown in Figure 7.19 demonstrates these concepts, as well as the use of the equality operator and the wrapper class's `equals` method. The output it produces is shown at the bottom of the figure.

Lines 2, 7, and 11 do not create a new object but simply assign `n2` the object's address that is stored in `n1`. Line 16 creates a new object even though the object `n1` created on line 15 stores the same value because the long form of the object-declaration syntax (which includes the keyword `new`) is used to create it.

```
1    Integer n1 = 20;
2    Integer n2 = 20; //a new object is not created, n2 is assigned n1
3    if(n1 == n2) System.out.println("n1 and n2 refer to the same object");
4    if(n1.equals(n2)) System.out.println("n1 & n2 contain the same value");
5
6    n1 = 30;
7    n2 = 30; //a new object is not created, n2 is assigned n1
8    if(n1 == n2) System.out.println("n1 & n2 refer to the same object");
9    if(n1.equals(n2)) System.out.println("n1 & n2 contain the same value");
10
11   n2 = n1; //a new object is not created, n2 is assigned n1
12   if(n1 == n2) System.out.println("n1 & n2 refer to the same object");;
13   if(n1.equals(n2)) System.out.println("n1 & n2 contain the same value");
14
15   n1 = 40;
16   n2 = new Integer(40); //a new object is created
17   if(n1 == n2) System.out.println("n1 & n2 refer to the same object");
18   if(n1.equals(n2)) System.out.println("n1 & n2 contain the same value");
```

Output produced:

n1 & n2 refer to the same object
n1 & n2 contain the same value
n1 & n2 refer to the same object
n1 & n2 contain the same value

```
n1 & n2 refer to the same object
n1 & n2 contain the same value
n1 & n2 contain the same value
```

Figure 7.18
Examples of when autoboxing creates new objects.

The *unboxing* feature of wrapper classes allows us to use the name of a numeric wrapper class object in arithmetic expressions. The following code fragment outputs the values *7* and *16*. The value stored in the object `n2` is unboxed from it on lines 2, 3, and 4. The unboxing fetches the value 7 from the object on lines 2 and 3, and the value 8 on line 4.

```
1  Integer n2 = 7;
2  int n3 = n2; //auto Unboxing of n2
3  n2++; //auto Unboxing of n2, incrementing, and Autoboxing the new value
4  n2 = n2 * 2; //auto Unboxing, multiplying, and Autoboxing the new value
5  System.out.println(n3 + " " + n2);
```

The location of wrapper class objects can be passed to and returned from a method using the same syntax used to pass any object's location to or from a method. The most common use of wrapper class objects is to pass a primitive value to a generic method that is expecting an object or to store a group of primitive values in a Java collection object. Generic methods and collections, and the role wrapper objects play in their use, will be discussed in Chapter 13.

7.6.3 Wrapper Class Constants

The six numeric wrapper classes all contain static data members named `MAX_VALUE`, `MIN_VALUE`, and `SIZE`. The values of a class's `MAX_VALUE` and `MIN_VALUE` constants are the maximum and minimum values that can be stored in the primitive numeric data member of an instance of that class. The value of the constant `SIZE` is the number of bits that make up the primitive data member's storage cell.

The following code fragment produces the output *127 -128 8*, which are the maximum and minimum values that can be wrapped in a `Byte` object, and the size of the object's data member (8 bits). These are the same values presented on the first row of Table 2.1, which specified the range and size of the primitive numeric types.

```
System.out.println(Byte.MAX_VALUE, + " " + Byte.MIN_VALUE, +
                   " " + Byte.SIZE);
```

7.6.4 The Character Wrapper Class

The wrapper class `Character` contains all of the methods, constants, and the autoboxing features of the numeric wrapper classes discussed previously in Section 7.6. Its value method, named `charValue`, returns the character stored inside the `Character` instance that invokes it. Naturally, the `Character` class's constants such as `MAX_VALUE`, `MIN_VALUE`, and `SIZE` store values relevant to primitive `char` values, and the class's methods operate on `Character` objects and are passed `char` parameters.

In addition to the analogous numeric wrapper class methods, the `Character` class contains two static methods named `toUpperCase` and `toLowerCase` that change the case of a character primitive passed to their `char`-type parameter. The `Character` class also contains seven other static methods described in Table 7.2 that return a Boolean value. These methods can be used to determine if the character passed to it is a digit or a letter, an uppercase or lowercase letter, or is Java whitespace. The methods are very useful in programs that process text information.

Figure 7.19 contains a program `ParseSentence` that demonstrates the use of four of the methods shown in Table 7.2 to determine the number of upper and lowercase letters, digits, and whitespace in an input sentence. A typical input sentence and the corresponding output the program produces is shown in Figures 7.20a and 7.20b, respectively.

The code inside the for loop that begins on line 14 of Figure 7.19 counts the number of occurrences of the four different types of characters contained in the sentence input on line 13. The `String` class's l`ength` method is invoked on line 14 to determine the number of characters in the sentence.

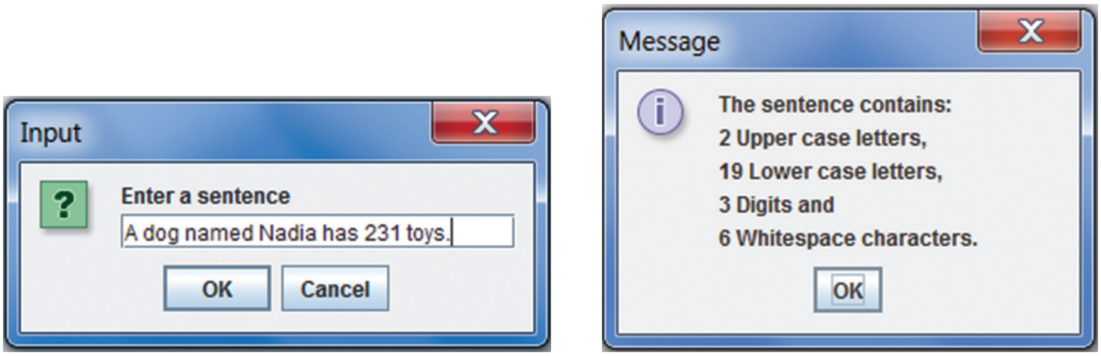
Each time through the loop, the loop variable is passed to the `String` class's `charAt` method to fetch the `i`th character from the sentence. Lines 17, 21, 25, and 29 use four of the character-testing methods presented in Table 7.2 to determine if the character returned from the `charAt` method is an upper- or lowercase letter, digit, or whitespace.

Table 7.2
The Character-Testing Methods in the Class Character

Method Name and Parameter List	Returns `true` if the Character Passed to its Parameter is:	Code Example
`isDigit(char c)`	a digit in the range 0 to 9	`Character.isDigit('4');` returns `true`
`isLetter(char c)`	an upper or lower case letter of the alphabet	`Character.isLetter('C');` returns `true`
`isLetterOrDigit(char c)`	an upper- or lowercase letter or digit (0-9)	`Character.isLetterORDigit('C');` returns `true` `Character.isLetterORDigit('6');` returns `true`
`isLowerCase(char c)`	a lowercase letter of the alphabet	`Character.isLowerCase('c');` returns `true`
`isSpaceChar(char c)`	a space character	`Character.isSpaceChar(' ');` returns `true`
`isUpperCase(char c)`	an uppercase letter of the alphabet	`Character.isUpperCase('B');` returns `true`
`isWhiteSpace(char c)`	Java defined white space (e.g., space, tab, or a new line character)	`Character.isWhiteSpace(' ');` returns `true`

```
import javax.swing.*;

public class ParseSentence
{
  public static void main(String[] args)
  {
    int upperCase = 0;
    int lowerCase = 0;
    int numeric = 0;
    int whitespace = 0;
    char c;

    String sentence = JOptionPane.showInputDialog("Enter a sentence");
    for(int i = 0; i < sentence.length(); i++)
    {
      c = sentence.charAt(i);
      if(Character.isUpperCase(c))
      {
        upperCase++;
      }
      else if(Character.isLowerCase(c))
      {
        lowerCase++;
      }
      else if(Character.isDigit(c))
      {
        numeric++;
      }
      else if(Character.isWhitespace(c))
      {
        whitespace++;
      }
    }
    JOptionPane.showMessageDialog(null, "The sentence contains:\n" +
                               upperCase + " Upper case letters,\n" +
                               owerCase + " Lower case letters,\n" +
                               numeric + " Digits and\n" +
                               whitespace + " Whitespace characters");
  }
}
```

Figure 7.19
The application **ParseSentence**.



(a) Input (b) Output

Figure 7.20
An input to the application **`ParseSentence`** and the resulting output.

7.7 AGGREGATION

Just as a group of primitive variables can be collected into an object by declaring them as data members in the object's class, instances of other types of objects can also be collected, or aggregated, into an object. We have already utilized this concept many times in this textbook. For example, lines 10 and 11 of Figure 7.12 indicate that a `String` and a `Color` object will be part of a `ParentSnowmanV2` object. As a result, the `ParentSnowmanV2` class would be considered an **aggregated** *class*.

Definition

An **aggregated** class is a class that includes in its data members instances of other classes.

Aggregating an object into a class is a simple task. As shown in Figure 7.21, which contains lines 10–11 and lines 20–28 of Figure 7.12, we simply declare a data member that can reference the object (lines 10 and 11) and assign the reference variable the address of an object (lines 11, 25, and 26).

```
10     private String name;
11     private Color hatColor= Color.BLACK;
 :
20     public ParentSnowmanV2(int intialX, int intialY, String name,
21                            Color hatColor)
22     {
23       x = intialX;
24       y = intialY;
25       this.name = name;
26       this.hatColor = hatColor;
27       snowmanCount++;
28     }
```

Figure 7.21
A code fragment from an aggregated class.

The difficult part of the aggregation shown in Figure 7.21 falls on the authors of the `String` and `Color` classes. They had to anticipate the operations that would be performed on aggregated strings or `Color` objects and provide methods to perform these operations. In the case of the `String` class, this includes operations such as comparing two strings, outputting a string to the console, and all of the other operations that were discussed in Section 7. 5. These methods are invoked by the code of the aggregated class to operate on the objects.

Properly anticipating the operations that will be performed on an aggregated object is a key component of the concept of aggregation. In addition, where possible, the signatures of these methods are standardized, as is the case for the `equals` method in the `String` and `Color` classes. Providing commonly used methods whose signatures are standardized facilitates the use of aggregation in our programs.

Instances of a class that we write can also be aggregated into other classes using the same syntax discussed above. (In this case, the burden of identifying the operations that will be performed on the aggregated objects and writing the classes falls on us.) This use of aggregation gives us the ability to extend the design concept of divide and conquer, used to divide complicated methods into smaller methods, to classes we write. The component objects of a large class (e.g., a `ParentSnowman` class) can be identified (e.g., a `Hat` object and a `Nose` object), and then the component classes can be written. Once written, instances of these classes can be aggregated into the larger class. In addition, instances of component classes can be aggregated into the classes of other programs just as instances of API classes are used in most Java programs.

The concept of aggregation gives us the ability to:

- use instances of existing objects in classes we write
- extend the design concept of divide and conquer to classes we write
- define a complicated object as an aggregation of component objects defined in other classes
- easily operate on aggregated component objects as separate entities

Figure 7.22 shows the UML representation of a class named `SnowmanV8` that aggregates a `Hat` and `String` object into it. The symbol for aggregation in a UML diagram is the diamond shown below the `SnowmanV8` class. The lines that connect it to the `Hat` and `String` classes indicate that the `SnowmanV8` class aggregates at least one `Hat` and one `String` object into it. The last two data members of the `SnowmanV8` class specify that one of each of these objects is aggregated into the class.

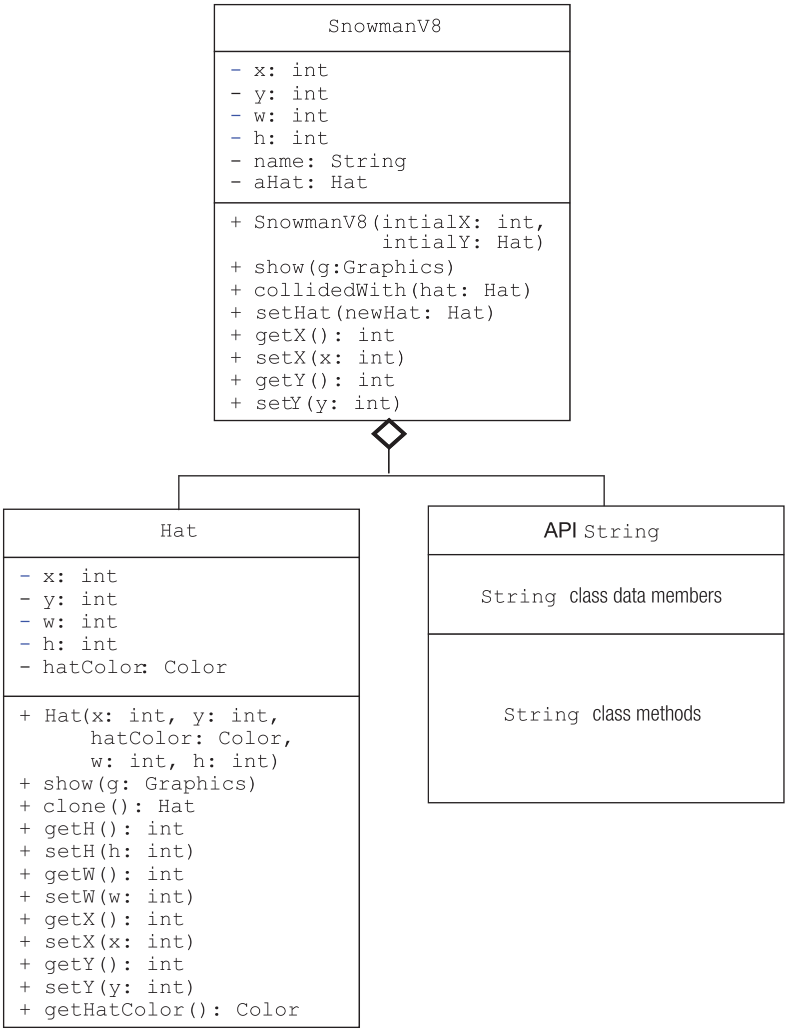


Figure 7.22
UML diagram of the **`SnowmanV8`** and the **`Hat`** classes.

During the design of the `Hat` class, it was anticipated that classes that aggregate hats will want to include hats of different colors and sizes. The last three parameters of the class's constructor were included in its parameter list to allow for this, as were its last three data members. The `Hat` class's `show` method will use these data members to draw a properly sized and colored hat. In addition, it has been anticipated that classes that aggregate `Hat` objects will want to clone them, so the class includes a clone method. Finally, it was assumed that once created, a hat's color would not be changed, so a `setHatColor` data member was not included in the `Hat` class.

Figure 7.23 shows the code of the class `Hat` that is specified in Figure 7.22. Lines 22 and 23 of the `show` method scale the hat top and its brim using the hat's specified width (`w`) and height (`h`). This unburdens the authors of all classes that aggregate hats from knowing how to scale a `Hat` object. That task is left to the hat specialist, which illustrates another advantage of aggregation.

In addition, they do not have to include the `Hat` class's data members and method in their class, making the aggregated class easier to code and understand. The clone method, lines 25–29, invokes the class's constructor in line 27 to create a clone of the `Hat` object that invokes it and returns the address of the newly created clone on line 28.

```
1    import java.awt.*;
2
3    public class Hat
4    {
5      private int x;
6      private int y;
7      private int w = 20;
8      private int h = 17;
9      private Color hatColor;
10
11     public Hat(int x, int y, Color hatColor, int w, int h)
12     {
13       this.x = x;
14       this.y = y;
15       this.hatColor = hatColor;
16       this.w = w;
17       this.h = h;
18     }
19     public void show(Graphics g)
20     {
21       g.setColor(hatColor);
22       g.fillRect(x + w/4, y, w/2, (int)(h*0.9));  // hat top
23       g.fillRect(x, y + (int)(h*0.9), w, (int)(h*0.2));  // brim
24     }
25     public Hat clone()
26     {
27       Hat theClone = new Hat(x, y, hatColor, w, h);
28       return theClone;
29     }
30     public int getW()
31     {
32       return w;
33     }
34     public int getH()
35     {
36       return h;
37     }
38     public int getX()
39     {
40       return x;
41     }
42     public void setX(int newX)
43     {
```

```
      x = newX;
    }
    public int getY()
    {
      return y;
    }
    public void setY(int newY)
    {
      y = newY;
    }
    public Color getHatColor()
    {
      return hatColor;
    }
  }
```

Figure 7.23
The class **Hat**.

Figure 7.24 shows the code of the class `SnowmanV8` that is specified in Figure 7.22. This class begins the aggregation of a `String` and a `Hat` object with the declaration of its last two data members on lines 9 and 10. Line 16 of the constructor completes the aggregation of the `String` object by creating a string and setting its address into the data member `name`.

The aggregation of the `Hat` object is completed on line 52 of the `setHat` method that assigns the address of a `Hat` object passed to its parameter on line 50 to the data member `aHat` declared on line 10. This means that until the `setHat` method is invoked, a `SnowmanV8` object should be drawn without a hat on his head. This is easily accomplished by including an `if` statement (line 22) in the class's `show` method that only invokes the `Hat` class's `show` method (line 26) when the data member `aHat` does contain its default value, `null`. This demonstrates another advantage of aggregation: aggregated objects can easily be treated as separate entities in a program. The snowman can be drawn with or without a hat.

Before the hat is drawn, it must be positioned on the snowman's head. This is accomplished by lines 24 and 25, which use the `Hat` class's `setX` and `setY` methods to set the (x, y) position of the hat above the snowman's head. The argument sent to these methods uses the `Hat` class's `setW` and `setH` methods to fetch the height and width of the hat, which are used to center the hat on the snowman's head.

Lines 38–49 of Figure 7.24 contain the code of the `SnowmanV8` class's `collidedWith` method specified in Figure 7.22. It detects a collision between the snowman that invokes it and the `Hat` object passed to its parameter. It uses the `Hat` class's `get` methods in the Boolean expression on lines 40 and 41 to decide if a collision has occurred.

```
import java.awt.*;

public class SnowmanV8
{
  private static int w = 40;
  private static int h = 77;
  private int x;
  private int y;
  private String name; //data members for aggregated objects
  private Hat aHat;

  public SnowmanV8(int intialX, int intialY)
  {
    x = intialX;
    y = intialY;
    name = "sm"; //aggregates a String object into this class
  }
  public void show(Graphics g)
  { int[] xPoly = {x + 20, x + 15, x + 25};
    int[] yPoly = {y + 8, y + 13, y + 13};

    if(aHat != null) //snowman has a hat
    {
      aHat.setX(x + w/2 - aHat.getW()/2); //locate the hat on head
      aHat.setY(y - aHat.getH());
      aHat.show(g); //draw the hat
    }
    g.setColor(Color.WHITE);
    g.fillOval(x + 10, y, 20, 20); // head
    g.fillOval(x, y + 20, 40, 40); // body
    g.setColor(Color.RED);
    g.fillPolygon(xPoly, yPoly, 3); // nose
    g.setColor(Color.BLACK);
    g.setFont(new Font("Arial", Font.BOLD, 16));
    g.drawString(name, x + 10, y + 45); // name
  }

  public boolean collidedWith(Hat hat)
  {
    if( !(x > hat.getX( ) + hat.getW() || x + w < hat.getX( ) ||
          y > hat.getY( ) + hat.getH() || y + h < hat.getY( )))
    {
      return true;
    }
    else
    {
      return false;
    }
  }
```

```
50    public void setHat(Hat newHat)
51    {
52      aHat = newHat; //aggregates a Hat object into this class
53    }
54    public int getX()
55    {
56      return x;
57    }
58    public void setX(int newX)
59    {
60      x = newX;
61    }
62    public int getY()
63    {
64      return y;
65    }
66    public void setY(int newY)
67    {
68      y = newY;
69    }
70  }
```

Figure 7.24
The class **SnowmanV8**.

Figure 7.25 presents the application `Aggregation`. When launched, it displays a hatless snowman and a hat rack containing six different hats, as shown in Figure 7.26a. The keyboard cursor-control keys are used to move the snowman to the hat he wishes to wear, perhaps the blue hat as shown in the upper right side of the figure (7.26b). The chosen hat is cloned and positioned on his head (Figure 7.26c), and follows him around the game board as shown on the lower right side of the figure (7.26d).

The application's `draw` method (lines 23–32) draws the hat rack on lines 25 and 26, and then its `for` loop invokes the hat class's `show` method to draw the six hats created on lines 12 to 17. The parameters sent to the `Hat` class's five-parameter constructor specify different locations, colors, and sizes for each hat.

Line 31 of the `draw` method invokes the `SnowmanV8` class's `show` method to draw the snowman that was created on line 8 on the game board. When the application is launched, the snowman's data member, `aHat`, declared on line 10 of the Figure 7.24, still contains its `null` default value. This causes the snowman to be drawn without a hat, as shown in Figure 7.26a, because the Boolean condition on line 22 of Figure 7.24 is `false`.

Every time a cursor key is struck, the code of the switch statement (lines 37–62) inside the `keyStruck` call back method moves the snowman two pixels left, right, up, or down. Then, the `if` statement (lines 66–69) uses the loop variable of the `for` loop to check each hat in the array `hats` to determine if the snowman has chosen (collided with) any of the hats on the hat rack.

When a hat is chosen it is cloned, and the clone is aggregated into the snowman object by invoking the hat class's `setHat` method and passing it the address of the cloned hat (line 68). The aggregation is accomplished by line 52 of Figure 7.24, which assigns the cloned hat's address (passed to the `setHat` method) the snowman's data member `aHat`. Because the data member is no longer `null`, lines 24–25 of Figure 7.24 position the aggregated hat centered and on top of the snowman's head at its current position, and line 26 draws the hat at that position.

```
1   import edu.sjcny.gpv1.*;
2   import java.awt.*;
3
4   public class Aggregation extends DrawableAdapter
5   { static Aggregation ge = new Aggregation();
6     static GameBoard gb = new GameBoard(ge, "Aggregation");
7     static Hat[] hats = new Hat[6];
8     static SnowmanV8 sm;
9
10    public static void main(String[] args)
11    {
12      hats[0] = new Hat(40, 100, Color.RED, 20, 17);
13      hats[1] = new Hat(120, 100, Color.ORANGE, 25, 21);
14      hats[2] = new Hat(200, 100, Color.YELLOW, 20, 17);
15      hats[3] = new Hat(280, 100, Color.GREEN, 40, 34);
16      hats[4] = new Hat(360, 100, Color.BLUE, 30, 25);
17      hats[5] = new Hat(440, 100, Color.MAGENTA, 35, 29);
18      sm = new SnowmanV8(250, 250);
19
20      showGameBoard(gb);
21    }
22
23    public void draw(Graphics g)
24    {
25      g.setColor(Color.BLACK); //the hat rack
26      g.fillRect(20, 95, 460, 5);
27      for(int i=0; i<hats.length; i++)
28      {
29        hats[i].show(g);
30      }
31      sm.show(g);
32    }
33    public void keyStruck(char key) //call back method
34    {
35      int newX, newY;
36
37      switch (key) //to move the snowman
```

```
      {
        case 'L':
        {
          newX = sm.getX() - 2;
          sm.setX(newX);
          break;
        }
        case 'R':
        {
          newX = sm.getX() + 2;
          sm.setX(newX);
          break;
        }
        case 'U':
        {
          newY = sm.getY() - 2;
          sm.setY(newY);
          break;
        }
        case 'D':
        {
          newY = sm.getY() + 2;
          sm.setY(newY);
        }
      }
      //acquiring a new Hat
      for(int i = 0; i<hats.length; i++)
      {
        if(sm.collidedWith(hats[i])) //a hat is chosen
        {
          sm.setHat(hats[i].clone()); //clone the hat and add it to sm
        }
      }
    }
}
```

Figure 7.25
The application **Aggregation**.

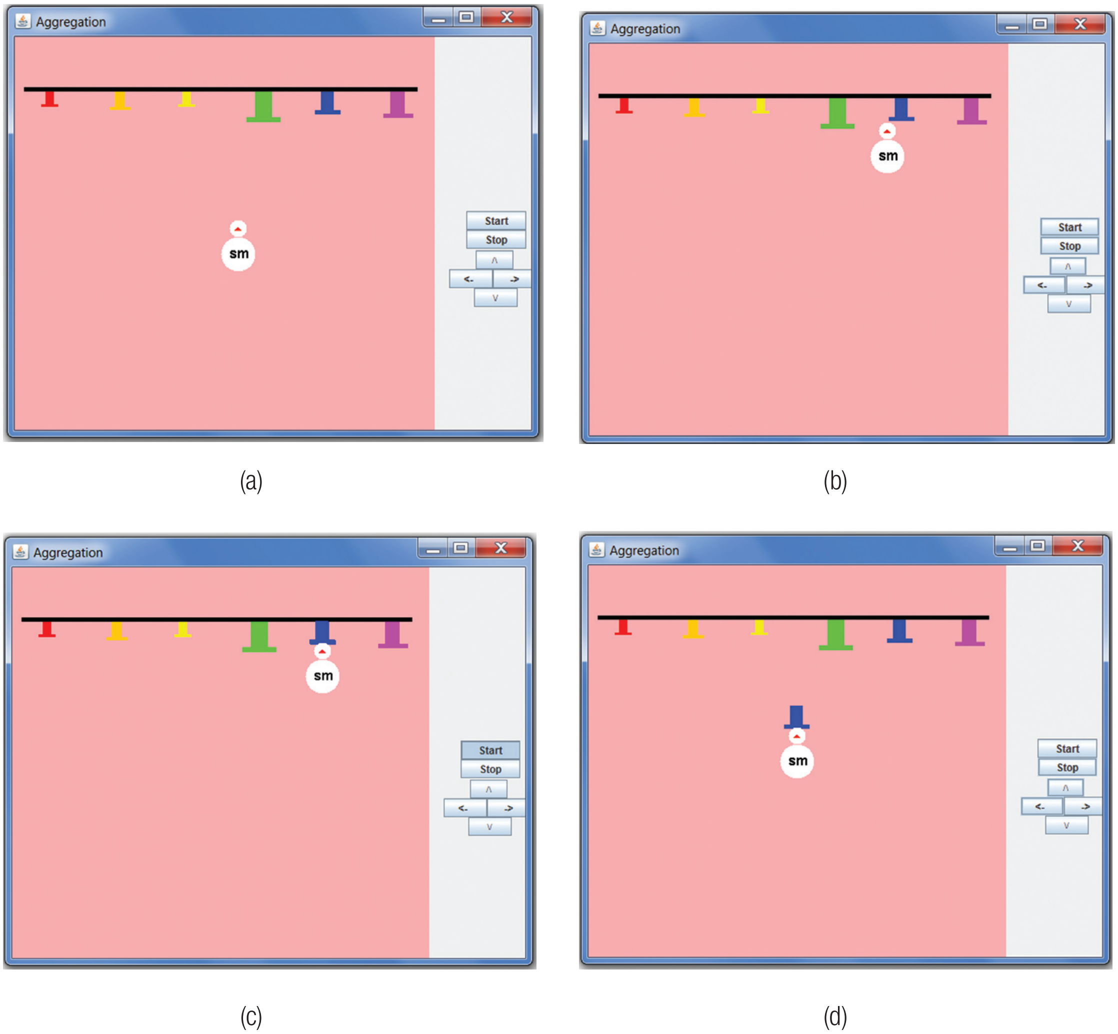


Figure 7.26
Output generated by the application **`Aggregation`**.

7.8 INNER CLASSES

An *inner class* is a class defined inside another class. Just as classes can contain data members and methods, they can contain other classes. The class that contains the inner class is called the *outer* class. Normally, a class is defined as an inner class only if the outer class will aggregate instances of the inner class. Consider the `Hat` class (Figure 7.23) discussed in the previous section. Because both the class `SnowmanV8` and the application `Aggregation` declared `Hat` objects, the `Hat` class was not coded inside the `SnowmanV8` class. Inner classes are most often used in Java programs that use a Graphical User Interface (GUI), also called a point-and-click interface. In this section, we will become familiar with the syntax of inner classes and the ability of the inner and outer classes to access each other's data members and methods.

Figure 7.27 presents the class `PhoneBook` that contains an inner class `PhoneNumbers`, which begins on line 28 and ends on line 48. Instances of the inner class contain a person's office, cell, and home phone numbers. These three strings are declared as aggregated data members of the inner class on lines 30–32, and the class's constructor assigns them input values (lines 36–40) when a `PhoneNumbers` object is created. The class's `show` method (lines 43–47) outputs the three phone numbers to the system console.

```
1   import javax.swing.*;
2
3   public class PhoneBook
4   {
5     private String[] name;
6     private PhoneNumbers[] numbers;
7
8     public PhoneBook() //a phone book has three listings
9     {
10      name = new String[3];
11      numbers = new PhoneNumbers[3];
12
13      for(int i = 0; i < name.length; i++)
14      {
15        name[i] = JOptionPane.showInputDialog("enter your name");
16        numbers[i] = new PhoneNumbers(i);
17      }
18    }
19    public void showAll()
20    {
21      for(int i = 0; i < name.length; i++)
22      {
23        System.out.println("\nName: " + name[i]);
24        numbers[i].show();
25      }
26    }
27
28    private class PhoneNumbers //an inner class
29    {
30      private String home;
31      private String cell;
32      private String office;
33
34      public PhoneNumbers(int i)
35      {
36        home = JOptionPane.showInputDialog("enter " + name[i] +
37                                           "'s HOME number");
38        cell = JOptionPane.showInputDialog("enter " + name[i] +
39                                           "'s CELL number");
40        office = JOptionPane.showInputDialog("enter " + name[i] +
41                                             "'s OFFICE number");
```

```
42        }
43        public void show()
44        {
45          System.out.println("PhoneNumbers: home:" + home +
46                                " cell:" + cell +  " office:" + office);
47        }
48      } //end of the inner class Phonenumbers
49    }
```

Figure 7.27
The **`class PhoneBook`** and its inner class **`PhoneNumbers`**.

The outer class `PhoneBook` declares two parallel arrays each containing three elements on lines 10 and 11 and assigns their addresses to the variables `name` and `numbers`, declared on lines 5 and 6. After creating the arrays, the `for` loop (lines 13–17) in the outer class's constructor completes the aggregation of the `String` and `PhoneNumbers` objects into the outer class by creating new instances of these classes and assigning their addresses to the elements of the parallel arrays `name` and `numbers` (lines 15 and 16).

NOTE *The code of an outer class can invoke a constructor in an inner class.*

Each time the inner class's constructor is invoked on line 16, it not only creates a new object, but it also accepts input into the data members of the newly created object on lines 36–41. The argument passed to this one-parameter constructor on line 16 is the loop variable declared on line 13. Lines 36, 38, and 40 of the constructor use this value to index into the outer class's `name` array, which causes the person's name that was just input on line 10 to become part of the prompts output by lines 36–41.

NOTE *The code of an inner class can access the data members of its outer class.*

The outer class's `showAll` method (lines 19–26) outputs all of the names and numbers in the two parallel arrays to the system console. Inside its `for` loop, line 23 outputs a person's name, and then line 24 invokes the inner class's `show` method to output that person's phone numbers.

NOTE *The code of an outer class can invoke the methods in an inner class.*

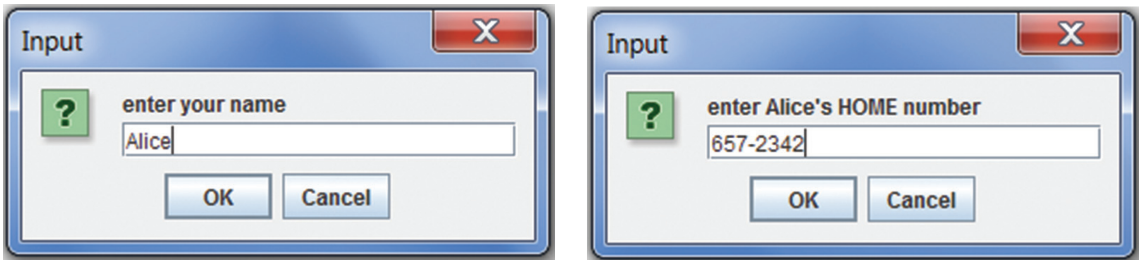
In general:

- the code of an inner and outer class can access each other's members (both data members and methods)
- an inner class is not visible outside of the outer class

The application `InnerClass`, shown in Figure 7.28, declares a `PhoneBook` object to store the names and phone numbers of a person's three best friends. Then, it invokes the class's `showAll` method to output the names and numbers input by the program user. Figure 7.29 shows two typical first inputs and a typical console output produced by the program.

```
public class InnerClass
{
  public static void main(String[] args)
 {
   PhoneBook bestFriends = new PhoneBook();
   bestFriends.showAll();
 }
}
```

Figure 7.28
The application **`InnerClass`**.



(a) (b)

Console Output:
Name: Alice
PhoneNumbers: home:657-2342 cell:574-8976 office:345-6589

Name: Tom
PhoneNumbers: home:367-4367 cell:754-3564 office:386-1212

Name: Annie
PhoneNumbers: home:456-4698 cell:765-8294 office:839-5623

Figure 7.29
Two typical inputs to the application **`InnerClass`** and a typical set of outputs.

7.9 PROCESSING LARGE NUMBERS

Occasionally, there is a need to process an integer that is larger or smaller than the maximum or minimum values that can be stored in the primitive type `long`. For example, the encryption used on the Internet involves processing prime numbers that contain 309 digits, which far exceeds the 19-digit maximum value that can be stored in the primitive type `long`. Similarly, we might need to process a real number that is larger or smaller than the maximum or minimum values of the primitive type `double` or require more than the 15 digits of precision the type `double` provides.

The Java API contains two classes that can be used to process numbers that are too large or too small to be stored in primitive types or that require more than 15 digits of precision; they are the `BigInteger` and `BigDecimal` classes. As their names imply, they can be used to process numbers beyond the range of the primitive integer and real types, and the `BigDecimal` can also provide a specified number of digits of precision. Objects in these classes, like `String` and wrapper class objects, are immutable.

Processing of objects in these classes is performed by using the methods that are part of these classes. For example, addition is performed by the method `add`, which adds the object that invoked it to the object passed to its parameter and returns a reference to an object that contains their sum. There are methods to perform all of the operations normally performed on primitive numeric types, including modulo arithmetic, methods that are analogous to the `Math` class's methods, and methods to perform other common operations on numbers such as finding the greatest common denominator, generating a prime number, and determining if a number is prime.

The `BigInteger` Class

Consider a program to generate a specified term of the Fibonacci sequence. The first two terms of this sequence, f1 and f2, are both defined as 1. Any other term in the series is defined as the sum of the two previous terms: fn = fn-2 + fn-1. From term 93 (f93) on, the values of the terms exceed the size of the primitive type `long`.

The application `FibonacciTerm`, shown in Figure 7.30, demonstrates the use of some of the constants and methods in the class `BigInteger`. The program calculates and outputs a specified term (greater than 2) of the sequence and identifies those terms that are larger than the maximum value of the primitive type `long`. A set of program inputs and corresponding outputs are shown in Figure 7.31.

Line 1 of Figure 7.30 imports the class `BigInteger` into the program, and lines 10–12 create three instances of the class. The first two are assigned the address of the class's static object that stores the value 1. There are two other static objects defined in the class, `TEN` and `ZERO`, which store the values 10 and 0, respectively. Line 12 uses the class's `valueOf` method that returns the address of a `BigInteger` object set to the value passed to its parameter. In this case, the maximum value of the primitive type `long` is passed to the method, which is a static constant in the wrapper class `Long`.

The `for` loop that begins on line 16 computes the terms of the sequence from 3 to the number input on line 14. Each time through the loop, the `add` method is used on line 19 to calculate the next term of the sequence and the address of the object containing the value returned from the method is assigned to the variable `fn`. Line 20 sets `fnMinus1` to the previous value of `fn`.

When the loop ends, line 22 uses the class's `toString` method to convert the calculated value stored in the object `fn` to a string before it is output. The conversion could have been coded as an implicit invocation of the `toString` method:

```
System.out.println("f" + n + " = " + fn);
```

Finally, the `BigInteger` class's `compareTo` method is used in the Boolean condition on line 23 to determine if the calculated term of the sequence is larger than the maximum value of a `long` type primitive. The integer value the method returns is interpreted in the same way as the integer value retuned from the `String` class's `compareTo` method.

```java
import java.math.BigInteger;
import javax.swing.*;

public class FibonacciTerm
{
  public static void main(String[] args)
  {
    int n;
    BigInteger temp;
    BigInteger fnMinus1 = BigInteger.ONE;
    BigInteger fn = BigInteger.ONE;
    BigInteger longMaxValue = BigInteger.valueOf(Long.MAX_VALUE);

    String s = JOptionPane.showInputDialog("enter the term number");
    n = Integer.parseInt(s);
    for(int i = 3; i <= n; i++)
    {
      temp = fn;
      fn = fnMinus1.add(fn);
      fnMinus1 = temp;
    }
    System.out.println("f" + n + " = " + fn.toString());
    if(fn.compareTo(longMaxValue) > 0)
    {
      System.out.println("Which EXCEEDS the maximum value of " +
                         "type long");
    }
    else
    {
      System.out.println("Which does NOT exceed the maximum value of " +
                         "type long");
    }
  }
}
```

Figure 7.30
The application **`FibonacciTerm`**.

Inputs to Three Executions of the Program:

92
93
300

Corresponding Outputs:

f92 = 7540113804746346429
Which does NOT exceed the maximum value of type `long`

f93 = 12200160415121876738
Which EXCEEDS the maximum value of type `long`

f300 = 222232244629420445529739893461909967206666939096499764990979600
Which EXCEEDS the maximum value of type `long`

Figure 7.31
Sample inputs to the application **`FibonacciTerm`** and the corresponding outputs they produce.

The `BigDecimal` Class

As previously mentioned, the `BigDecimal` class can be used to represent real values to a specified precision. The code fragment shown in Figure 7.32 computes and rounds up the number 176 divided by 7 to a precision of 19, 18, 17, and 16 digits. The resulting `BigDecimal` object, returned from the three-parameter version of the class's `divide` method, is output at the bottom of the figure using an implicit invocation of the `BigInteger` class's `toString` method. The `divide` method divides the object that invoked it by the first argument sent to it.

The second argument sent to the `divide` method specifies the precision of the computed value. The third argument specifies the rounding mode used to determine the rightmost digit of precision. This can either be an integer or a constant defined in the class `RoundingMode`. The class's constant HALF_UP, used in Figure 7.32, is used to perform the conventional upward rounding.

```
BigDecimal one76 = BigDecimal.valueOf(176);
BigDecimal seven = BigDecimal.valueOf(7);

System.out.println(one76.divide(seven, 19, RoundingMode.HALF_UP));
System.out.println(one76.divide(seven, 18, RoundingMode.HALF_UP));
System.out.println(one76.divide(seven, 17, RoundingMode.HALF_UP));
System.out.println(one76.divide(seven, 16, RoundingMode.HALF_UP));
```

Output:

25.1428571428571428571
25.142857142857142857
25.14285714285714286
25.1428571428571429

Figure 7.32
Use of the **`BigDecimal`** class's divide method.

7.10 ENUMERATED TYPES

Enumeration is the process of defining a new type and specifying, or enumerating, a finite set of values that instances of the type can assume. The use of enumeration can make our programs more readable. Java supports enumerated types. The following enumeration statement defines the enumerated type `Team` and specifies that there are three values in its set of values: `Yankees`, `Braves`, and `Giants`.

```
// Declaration of an enumerated type
   enum Team {Yankees, Braves, Giants}
```

An enumeration statement is coded at the class level; it cannot be coded inside (local to) a method. The values that appear in the statement can be any valid identifier, which implies that they cannot be strings or primitive literals. The following declarations are invalid:

```
// Invalid enumeration statements: values not identifier names
enum Team {"Yankees", "Braves", "Giants"} //can't contain quotes
enum ID {NY Yankees, Atlanta Braves, SF Giants} //spaces not allowed
```

An enumeration statement can also be written in a separate Java file whose name is the same as the statement's enumerated type name; e.g., `Team`.

At an abstract level, the identifiers can be thought of as static constants in a class whose name is the enumerated type name (e.g., `Team`) and whose values are the identifier names. In this abstract view, the enumerated values are analogous to the static `double` constant `PI` in the `Math` class whose value is the double 3.141592653589793. The following code fragment outputs *3.141592653589793 Yankees* to the system console:

```
// Accessing the values of enumerated types
enum Team {Yankees, Braves, Giants}

System.out.println(Math.PI + " " + Team.Yankees);
```

Continuing the analogy, just as the numeric constant `Math.PI` can be assigned to a variable of its type (i.e., `double`), an enumerated constant can be assigned to a variable of its type (e.g. `Team`). The following code fragment also outputs *3.141592653589793 Yankees* to the system console:

```
// Assigning an enumerated type value
enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Yankees;
double valueOfPi = Math.PI;
System.out.println(valueOfPi + " " + myTeam);
```

For the most part, considering the identifiers in an enumeration statement to be analogous to static constants is consistent with the Java syntax of enumerations. An exception to this is when an enumerated type variable is used in a switch statement after the keyword `switch`. In this context, the enumerated type name is not used to qualify the identifier coded as the choice value after the keyword `case`. In the code fragment shown in Figure 7.33, when an enumerated value such as Yankees is used within the switch statement, the type name `Team` is not used to qualify the identifier, as shown on line 6. When the identifier is used in other statements (e.g., line 7 of Figure 7.33), the type qualifier is used.

```java
// Using enumerated types in switch statements
enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Yankees;
switch (myTeam)
{
  case Yankees: // no type name qualifier used here
  System.out.println(myTeam + " " + Team.Yankees);
}
```

Figure 7.33
Syntax of enumerated types in a **switch** statement.

The API interface `Enum` defines a set of methods that implement operations commonly performed on enumerated types. When discussing them, it useful to move away from the abstract view of an enumeration's identifiers being static constants to a more concrete view of them. While it is true that the enumerated type is a class, it is a special kind of class. In addition, the identifiers are not static constants in the class, but rather each identifier is a static reference variable that refers to an instance of the class.

As such, like static constants, we access the contents of these reference variables by the name of the identifier proceed by the name of the class followed by a dot (just as we have been doing). Figure 7.34 shows the three objects created by the enumeration shown below, the variables that reference them, and the contents of the variable `myTeam` after the assignment statement is executed:

```java
// Shallow copy of an enumerated object

enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Yankees; //Shallow copy
```

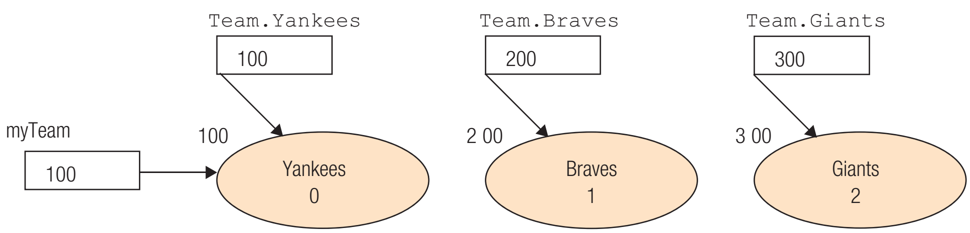


Figure 7.34
Three objects of the enumeration **Team**.

As shown in Figure 7.34, each object has an ordinal value associated with it, which always begins with zero. The values are assigned sequentially to the identifiers in the order (left to right) in which they appear in the enumeration statement. The `Enum` class's method `ordinal` returns the ordinal value assigned to the object that invoked it. The code fragment below outputs 1 to the system console.

```
// Invocations of the Team class' ordinal method
enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Braves
System.out.println(myTeam.ordinal());
```

The `ordinal` method can also be used with an enumerated type object in the same way any other method is used with an instance of its class. For example, the statement

```
System.out.println ( "The ordinal value of " + Team.Giants +
                     " is " + Team.Giants.ordinal() );
```

results in the console output *The ordinal value of Giants is 2* .

NOTE *The ordinal values of an enumerated type always begin with zero.*

The methods in the class `Enum`, four of which are shown in Table 7.3, operate on an enumerated type's objects. The class's `toString` method returns the name of the object's identifier converted to a string. The code fragment below outputs *Yankees Yankees* using an explicit and implicit invocation of the method:

```
// Invocations of the Enum class's toString method
enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Yankees
System.out.println(Team.Yankees.toString() + " " + myTeam.Yankees);
```

Table 7.3
Commonly Used Methods in the Class Enum

Method Name and Parameter List	Returned Type	Description
`compareTo(EnumType enum2)`	`int`	`diff = enum1.compareTo(enum2)` Returns the difference between the ordinal values of `enum1` and `enum2`, positive for `emum1 > enum2`
`equals(EnumType enum2)`	`boolean`	`equal = enum1.equals(enum2)` Returns `true` when the ordinal values of `enum1` and `enum2` are equal, otherwise `false`
`ordinal()`	`int`	`value = enum1.ordinal()` Returns the ordinal values of `enum1`
`toString()`	`String`	`value = enum1.toString()` Returns the value of `enum1`

The `Enum` class's `compareTo` method is an implementation of the method defined in the interface `Comparable`. The comparison it makes is based on the ordinal values associated with the object that invoked the method and the argument passed to it. The following code fragment outputs -2 because the identifier `Yankees`' ordinal value (0) is two less than the identifier `Giants`' value (2).

```
// Invocations of the Team class' compareTo method
enum Team {Yankees, Braves, Giants}

Team myTeam = Team.Yankees
System.out.println(myTeam.compareTo(Team.Giants));
```

The `Enum` class's method `equals` returns `true` when the object that invoked it has the same ordinal value as the object passed to its parameter.

7.11 CHAPTER SUMMARY

Java classes consist of data members and methods that operate on the data, and an object is an instance of a class. When a data member is declared to be static, all instances of the class share the variable. Often, a static variable is incremented inside a class's constructor to keep track of the number of objects that have been created.

It is good programming practice to write a complicated method as several simpler methods that it can invoke. Usually, the simpler methods are given private access to prevent methods that are not part of the class from invoking them. Methods can invoke private methods that are part of their class by simply coding the method's name and an argument list.

The object addresses stored in two reference variables can be compared using the relational operators, and they can be copied using the assignment operator. These are referred to as a shallow comparison and a shallow copy, respectively. After a shallow copy, two reference variables refer to the same object. To compare or copy objects, we first need to define which of their data members will be compared or which will be copied. Comparing and copying the data members of two objects is referred to as deep comparisons and deep copies. In both cases, a method has to be written to perform these operations.

Deep comparison methods return a Boolean value, and it is good programming practice for the methods to be named `compareTo`, as defined in the API interface `Comparable`, or named `equals` when the comparison is performed to determine equality. The `String` class contains deep comparison methods with these names. Deep copy methods either copy all of the data members from one object to another or copy all of the data members into a newly created object called a clone and return the address of the clone object. The names of deep copy methods ordinarily contain the word "copy," such as the `arraycopy` method in the `System` class, and clone methods are usually named `clone`.

In addition to deep comparison methods, the `String` class contains methods to perform common operations on `String` objects. These include locating and fetching substrings (the methods `indexOf` and `substring`), changing the case of a string (`toUpperCase` and `toLowerCase`), replacing a part of a string with another string (`replaceFirst` and `replaceAll`), and determining if a string begins or ends with a particular string (`startsWith` and `endsWith`). In addition, its `split` method can be used to place substrings of a string separated by a designated delimiter, usually white space, into the elements of the array it creates and returns. The substrings are called tokens, and the process is called tokenizing the string.

Autoboxing is the automatic construction of a wrapper class object without having to explicitly invoke the class' constructor, and auto-unboxing is the process of fetching the primitive value stored in a wrapper class object without having to invoke a `get` method. This feature can be used to pass a primitive value to a wrapper-class parameter or to assign a retuned wrapper class object to a primitive variable. The `Character` wrapper class contains a variety of methods used to process characters such as determining if a character is a letter or a digit, if it is lower or upper case, or if it is white space. The numeric wrapper classes contain static constants whose values are the maximum and minimum numeric values that they can wrap. The methods in the classes `BigInteger` and `BigDecimal` give us the ability to process numbers whose absolute value is too large to be stored in primitive types or that require more than 15 digits of precision. Objects in these classes, like `String` and wrapper class objects, are immutable.

Aggregation is the process of declaring a data member of a class to be a reference to an object. This permits us to define a complex object as an aggregate of simpler component objects, extending the concepts of reusable code and divide and conquer to the design of classes. A class, called an inner class, can also be defined within another class called an outer class. The inner class can access the data members defined in the outer class and vice versa. The outer class can create instances of the inner class and invoke its methods. Enumeration is the process of defining a type and specifying, or enumerating, the values that instances of the type can assume. The use of enumeration can make our programs more readable.

Knowledge Exercises

1. True or false:

a) Static data members allow one storage cell to be shared among all instances of their class.

b) A method cannot invoke another method in its class.

c) Methods that are declared `private` can be invoked by other methods within their own class.

d) A deep comparison determines if two reference variables refer to the same object.

e) To make a clone of an object, we make a shallow copy of it.

f) The variables `s1` and `s2` refer to two different objects. After I make a shallow copy of `s2` into `s1`, I have two identical objects.

g) Autoboxing constructs a wrapper class object without explicitly invoking the class's constructor.

h) An outer class can invoke the methods of its inner class.

2. Consider the class `Student` shown in Figure 7.4.

a) Why is its data member `studentCount` declared to be static?

b) Why is the class's `get` method declared to be a static method?

c) Write the statement to output the variable `studentCount` to the system console.

d) Why does the class not contain a public method named `setSudentCount`?

3. Explain the difference between a deep and a shallow comparison.

4. Explain the difference between a deep and a shallow copy.

5. Explain the difference between a deep copy and a clone.

6. How does the method `equals` in the API `Object` class differ from the `equals` method in the `String` class?

7. Write a code fragment to output *Two Objects* to the system console if the variable `s1` and `s2` refer to two different objects.

8. Write a method that could be added to the class `Student`, shown in Figure 7.5, which would clone the `Student` object that invoked it.

9. What is output by these statements, assuming the following declarations have been made:

```
String s1 = "Computers rock"; String s2 = "Hello world";
```

a) `System.out.println(s2.indexOf("world"));`

b) `System.out.println(s1.substring(10));`

c) `System.out.println(s2.replaceFirst("world", "everyone"));`

d) `System.out.println(s2.starts("world"));`

e)
```
String[] s = s2.split();
System.out.println(s[1]);
```

f)
```
if(s1.equalsIgnoreCase("Computers Rock")
{
System.out.print("True - these are the same.");
}
else
{
System.out.print("False - they are not the same.");
}
```

10. Give the code to:
    - **(a)** create an instance of the wrapper class `Integer` that contains the value 20 without explicitly invoking the class's constructor
    - **(b)** set the integer variable `age` to the value stored in the `Integer` instance `number`
    - **(c)** output the maximum and minimum values that can be stored in a primitive variable declared to be type `long` to the system console
    - **(d)** output `true` to the system console if the character contained in the variable `aCharacter` is white space or a digit

11. Define aggregation in the context of a Java class.

12. Give three advantages of using aggregation in the classes we design.

13. How would you represent an integer in your program that was larger than the maximum value of the primitive type `long`?

14. How would you input an integer to your program that was larger than the maximum value of the primitive type `long`?

15. How would you double the integer discussed in Exercise 13?

16. Give the code to define two `BigInteger` objects initialized to 1,234,567,890,123,456 and 9,876,543,210,654,321, multiply the numbers, and output the result.

17. Assume an enumerated type `CarColor` has been declared as:

    `enum CarColor` {RED, WHITE, SILVER, BLACK, BLUE}

    a) What is the ordinal value of SILVER?

    b) Write a statement to create a variable, `favoriteColor`, of type `CarColor` and set it to BLUE.

    c) Write a statement to output the `favoriteColor` and its ordinal value to the system console.

18. Give a sequence of code that uses the `StringTokenizer` class methods to fetch integers from this string and outputs their sum.

```
String numbers = "213 45434   -4537  34 65456  81"
```

Programming Exercises

1. Add a deep comparison method and a clone method to the class shown in Figure 7.4. The deep comparison method should return 0 when two instances of the class are equal (i.e., both objects' id numbers and GPA are equal). Then write an application that declares an instance of the class named `s1` and two other instances named `s2` and `s3`. The object `s2` should have the same id as `s1`, and `s3` should have the same GPA as `s1`. Output the three objects to the system console followed by *s1 equals s2* or *s1 equals s3*, as determined by two invocations of the deep comparison method. Then clone `s1`, store the returned address in `s2`, and repeat all of the output.

2. Write a program that accepts an arithmetic expression that does not contain parentheses and verifies that is correctly written. It is correctly written if each of the operators in the expression is between two operands preceded and followed by a space: for example 2 * 3 + 5. Output a correct expression to the system console. Otherwise, output the expression up to the point were the first error was detected, then a caret (Shift 6 key stroke), and then the remainder of the math expression. The second line of output, in both cases, should be the number of tokens (operators and operands) that were in the original expression. (Use the `split` method in the `String` class.)

3. Modify the class `PhoneBook` shown in Figure 7.27 so each of the three listings in a `PhoneBook` instance will also have an address consisting of a street, city, state, and zip code. To accomplish this, add another inner class named `Address` to the class `Phonebook`. The new information should be input in a similar way to that of the phone numbers. To verify your modifications to the class, write an application that declares an instance of a `PhoneBook`, accepts user inputs, then outputs the entire phonebook.

4. Write a program that calculates and outputs the sum of the integers from 1 to 10,000,000,000. Hint: the sum is (10,000,000,000 * 10,000,000,001) / 2, which is not equal to 3,883,139,820,726,120,960.

5. Write a program to multiply two real numbers of any size and precision input via a dialog box and output their product with seven digits of precision, rounded up.

Enrichment

1. Investigate and learn who was the first to discover that the sum of the integers from 1 to n is (n * (n +1)) /2 and the circumstances under which he discovered it.

2. Explore the Fibonacci sequence to discover its presence in art, architecture, music, and nature and investigate the relationship of the Fibonacci sequence to the Golden ratio.

3. Research the Fibonacci searching algorithm.

CHAPTER 8

INHERITANCE

In this chapter

In this chapter, we expand our knowledge of object oriented programming into the advanced topics of inheritance, polymorphism, interfaces and adapter classes, and the serialization of objects. These OOP concepts can significantly reduce the time and effort required to design and develop a software product.

We will learn that Java supports two forms of inheritance, chain inheritance and multiple child inheritance. Both of these allow us to rapidly create a class from a similar existing class and to easily expand and/or modify the new class to adapt it to the requirements of a particular project. When used as a design tool, inheritance not only reduces the cost of a software product under development, but it also reduces the cost of future products by increasing the reusability of the classes we write. Using inheritance, application-dependent parts of a method can be written in a way that they can invoke methods that implement the yet-to-be-determined requirements of future products.

Polymorphism is another fundamental characteristic of OOP, which allows things to exist in many ("poly") different forms ("morph"), such as when one array references many different types of objects or when one invocation morphs itself into an invocation of a method appropriate to a particular launch of a program. Polymorphism also allows us to pass different types of objects to one type of parameter.

Interfaces allow us to define the signature and functionality of related methods without having to implement them, and adaptor classes facilitate the use of interfaces. Using object serialization, we can easily save a collection of objects to a disk file and reuse the objects in a future launch of the same or different program.

After successfully completing this chapter you should:

- Understand the advantages, terminology, syntax, and importance of using inheritance
- Know how to implement a child class that inherits data members and methods from an existing parent class using the `extends` clause
- Be able to invoke and modify an inherited method and expand inherited methods and data members
- Be able to distinguish between overriding and overloading methods
- Know how to use inheritance as part of the design process
- Understand the processes that Java uses to locate a method at translation and run time
- Be able to comprehend and use polymorphism and polymorphic arrays
- Understand abstract classes, interfaces, and adapter classes and how to implement and use them
- Write and read a group of objects to and from a disk file

8.1 THE CONCEPT OF INHERITANCE

Definition

Inheritance is an OPP programming concept in which new classes that contain all of the data members and methods of an existing class can be efficiently created and then expanded and/or modified. The concept of inheritance implies that two classes have a relationship with each other in which one class, called a *child* class, inherits attributes from the other class, called the *parent* class.

The concept of **inheritance** is fundamental to object oriented programming. If properly used, it can greatly reduce the time and effort required to develop a software product. For example, suppose that one of the classes specified during the design of a new program is similar to an existing class in that the existing class contains many of the data members and methods listed in the new class's UML diagram. The best way to develop the new class would be to simply add the existing class to the program and modify and/or add to it. One approach to this would be to copy the source code of the existing class, paste the code into a new class, and then modify the copied code. A better approach would be to use the concept of inheritance.

Although we may have been importing the existing class into our programs for many years, we may not have its source code. For example, the Java API does not contain the source code of any of the classes included in it. Rather, it contains the classes' translated byte codes. This fact would eliminate the copy and paste alternative but not the inheritance alternative. To use the concept of inheritance, we only need the byte codes of the existing class. In addition, software engineering studies reveal that copying, pasting, and modifying code that we did not write can be more time consuming than a completely independent development of a new class. Inheritance allows us to modify an existing class that was added to a new program in a way that does not introduce the errors that are associated with the copy-paste-modify alternative.

Even if there are no existing classes that contain many of the data members and methods of the classes specified for a new program, the time and cost to develop the program can still be reduced using the concept of inheritance during the design process. Finally, the use of inheritance in the design of our programs also makes them easier to read and intuitively easier to understand because the classes we write better model the real world. Because we know that children inherit attributes from their parents, it is intuitive that a game's new `ChildSuperHero` class would inherit attributes from an existing game's `ParentSuperHero` class. The attributes inherited by the `ChildSuper-Hero` class would be the data members and the methods of the `ParentSuperHero` class, which could then be efficiently expanded or modified to model the child super hero.

In summary:

- Inheritance reduces the cost and time to develop a software product
- Inheritance is the best way to incorporate and then morph an existing class into a new class
- We do not need the source code of the existing class, only its byte code translation, to utilize the concepts of inheritance
- Inheritance is used in the OOP program-design process even when there are no existing classes to be morphed and incorporated into the program

8.2 THE UML DIAGRAMS AND LANGUAGE OF INHERITANCE

Inheritance introduces a new feature into UML diagrams and has a set of terms associated with it. An early understanding of this feature and the jargon of inheritance is fundamental to the remainder of the material in this chapter.

Inheritance UML Diagrams

Figure 8.1 shows the UML diagram of the classes `RowBoat` and `SailBoat`. The `RowBoat` class's UML diagram contains five data members and twelve methods, and the UML diagram of the `SailBoat` contains no data members and one constructor method. Instances of the class `SailBoat` will actually contain more data members and methods than those listed in the class's UML diagram because the upward-pointing arrow in the center of the figure indicates that the `SailBoat` class *inherits from* the `RowBoat` class. As noted next to the arrow in the figure, the Java keyword used to establish this relationship is **`extends`**.

The Parent-Child Relationship

The concept of inheritance implies that two classes have a relationship with each other in which one class, called a *child* class, inherits attributes from the other class, called the *parent* class. The attributes inherited are all of the data members and all of the methods of the parent class *except* for the parent class's constructors. As indicated on the left side of Figure 8.1, there are two other pairs of terms that are used in the literature to refer to the parent and child class. The parent class is sometimes called the *super class*, and then the child class is called the *subclass*. Alternately, the term *base class* can be substituted for parent class, in which case the child class is referred to as a

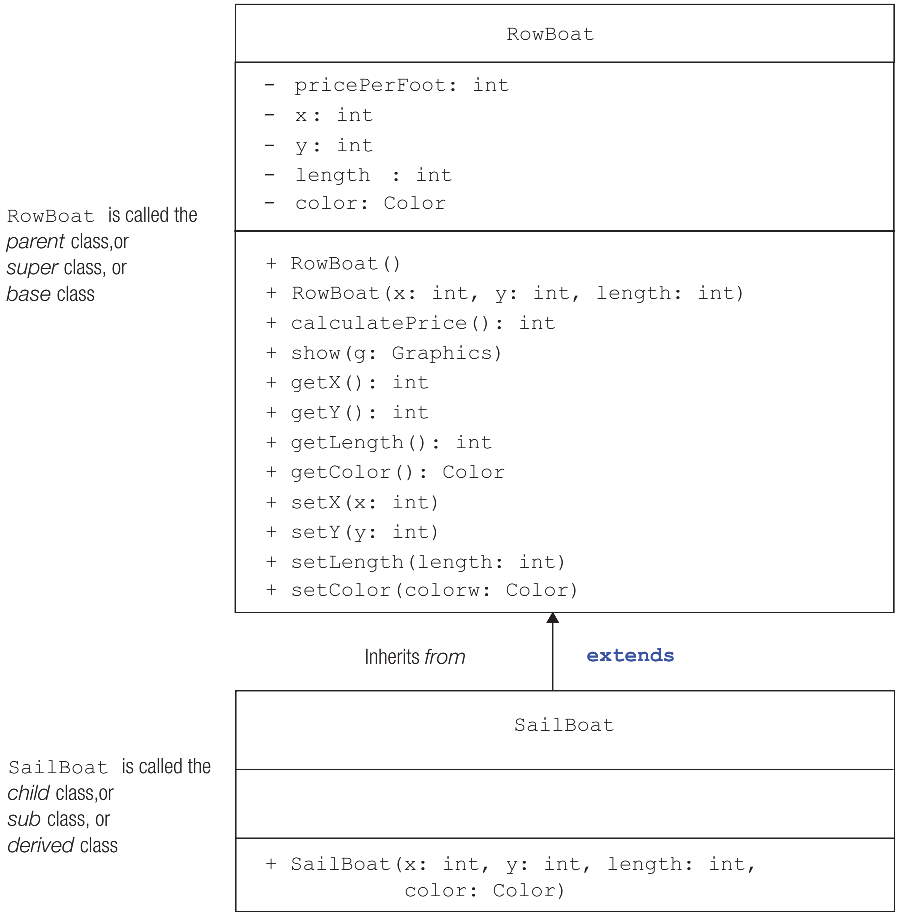


Figure 8.1
The class **`RowBoat`** and the class **`SailBoat`**.

derived class. These three pairs of terms: *parent* class-*child* class, *super* class-*sub* class, and *base* class-*derived* class should not be unpaired and intermixed.

Because a child class inherits all of the data members and methods of the parent class and contains all of the data members and methods specified in its own UML diagram, a child class's data members and methods add to, or *extend*, a parent class. As specified in Figure 8.1, all instances of the classes `SailBoat` and `RowBoat` will have five data members, but the `SailBoat` class extends the complement of methods that can operate on a `SailBoat` object from 12 to 13.

NOTE *Inheritance does not work in reverse. Parents do not inherit the data members and methods added to the child class.*

Establishing the Parent-Child Relationship

When a child class is implemented, the parent-child relationship is established by the inclusion of an **`extends`** *clause* on the right side of the class's heading. This clause begins with the keyword **`extends`** followed by the name of the parent class. For example:

```
public class SailBoat extends RowBoat
```

NOTE

To establish an inheritance relationship, an `extends` *clause is included at the end of the class statement of the child class:*

```
public class ChildClassName extends ParentClassName
```

Forms of Inheritance

A class can only extend or inherit from *one* class. In *chain inheritance* a child class can be the parent of another class. This is supported in Java, as is the ability for several child classes to inherit from the same parent class. These two forms of inheritance are illustrated in the upper-left and upper-right portions of Figure 8.2, respectively.

Multiple inheritance, the concept that a child class can inherit attributes from *more than one* parent, is not supported in Java. Multiple inheritance is depicted in the lower-right portion of Figure 8.2.

We say that class `B` on the top left side of Figure 8.2 *directly* inherits from class `A`, as do classes `B`, `C`, and `D` at the top right side of the figure. A class that directly inherits from another class extends it by including an `extends` clause in its heading. We say that classes `C` and `D` on the top left side of Figure 8.2 *indirectly* inherit from class `A` because they either inherit from a class that directly inherits from class `A` (in `C`'s case) or directly inherit from a class that indirectly inherit from class `A` (in `D`'s case).

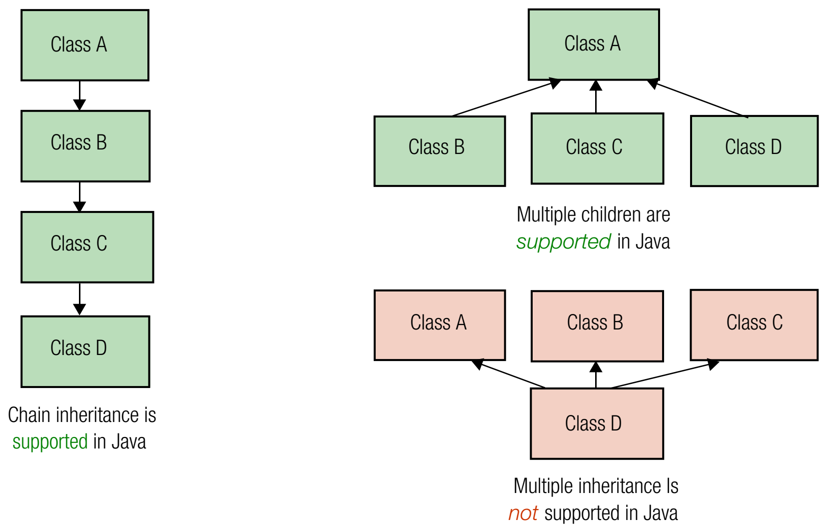


Figure 8.2
Various forms of inheritance.

8.3 IMPLEMENTING INHERITANCE

In addition to the keyword `extends`, which is used to indicate that a child class inherits from a parent class, there are other keywords and concepts that are associated with inheritance. Most of these are used in the implementation of a child class, but an understanding of some of these keywords, such as **final** and **abstract**, and the concepts associated with them, apply to the implementation of a parent class. In the remainder of this section, we will discuss the keywords and concepts of inheritance that apply to the implementation of a child class; we will discuss the relevant parent class issues in Section 8.4.

Figure 8.3 presents the code of the class `RowBoat` specified in the top portion of Figure 8.1. The class does not utilize any of the concepts of inheritance presented in this chapter. Nevertheless, it can become a parent class if another class extends it. This class will be the parent of child classes used in parts of this chapter to demonstrate the basic of concepts inheritance and the use of the keywords associated with these concepts.

```
import java.awt.*;
public class RowBoat
{
  private static int PRICE_PER_FOOT = 10;
  private int x, y, length;
  private Color color = Color.GREEN;

  public RowBoat()
  {

  }
  public RowBoat(int x, int y, int length)
  {
    this.x = x;
    this.y = y;
    this.length = length;
  }
  public int calculatePrice()
  {
    return length * PRICE_PER_FOOT;
  }
  public void show(Graphics g)
  {
    int[] xBoat = {x , x + length, x + 6 * length/7,  x + length/14};
    int[] yBoat = {y,  y, y + length / 7, y + length / 7};
    int price = calculatePrice();
    g.setColor(color); //draw the Boat
    g.fillPolygon(xBoat, yBoat, xBoat.length);
    g.setColor(Color.BLACK); //draw the boat's price in black

    g.setFont(new Font("Arial", Font.BOLD, 16));
    g.drawString("$" + String.valueOf(price), x + 10, y + 16);
  }
  public int getX()
  {
    return x;
```

```
36     }
37     public int getY()
38     {
39       return y;
40     }
41     public int getLength()
42     {
43       return length;
44     }
45     public Color getColor()
46     {
47       return color;
48     }
49     public void setX(int x)
50     {
51       this.x = x;
52     }
53       public void setY(int y)
54     {
55       this.y = y;
56     }
57     public void setLength(int length)
58     {
59       this.length = length;
60     }
61       public void setColor(Color color)
62     {
63       this.color = color;
64     }
65   }
```

Figure 8.3
The class **RowBoat**.

The `RowBoat` class's three-parameter constructor (lines 12–17 of Figure 8.3) can be used to create and position a rowboat on the game board at the (x, y) location passed to its first two parameters. The value passed to the constructor's third parameter is the size (length) of the rowboat. The `x`, `y`, and `length` data members of a rowboat created with the class's default constructor (lines 8–11) would retain the default value of the type `int`: 0. Because the constructor does not include a parameter to specify the color of the boat, it would default to green (line 6).

The `show` method (lines 22–32) draws a rowboat on the game board at its current (x, y) location and then draws the price of the boat on it (lines 29–31). The price is returned from the invocation of the class's `calculatePrice` method on line 26. Line 20 of this method multiplies the length of the boat by the static variable `PRICE_PER_FOOT` to determine the price of the boat.

The application `ShowTwoRowBoats` shown in Figure 8.4 creates two rowboats of lengths 200 and 150 feet (lines 7-8) and displays them on the game board (lines 16-17) as shown in Figure 8.5.

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class ShowTwoRowBoats extends DrawableAdapter
{ static ShowTwoRowBoats ge = new ShowTwoRowBoats();
  static GameBoard gb = new GameBoard(ge, "Show Two Row Boats");
  static RowBoat rb1 = new RowBoat(30, 150, 200);
  static RowBoat rb2 = new RowBoat(30, 250, 150);

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }
  public void draw(Graphics g)
  {
    rb1.show(g);
    rb2.show(g);
  }
}
```

Figure 8.4
The application **`ShowTwoRowBoats`**.

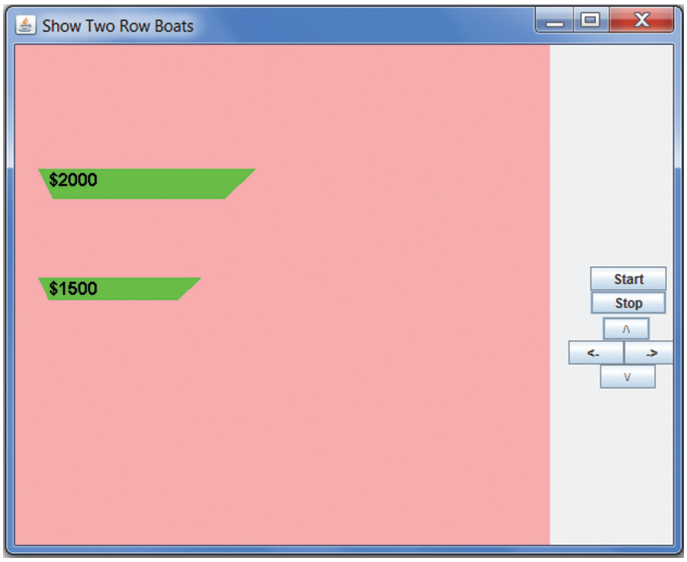


Figure 8.5
The output produced by the application **`ShowTwoRowBoats`**.

8.3.1 Constructors and Inherited Method Invocations

The class `SailBoat` is specified in the bottom portion of Figure 8.1. Sailboats use rowboats for their hull and are delivered without a mast and sail. As delivered, they actually are rowboats, except for the fact that the purchaser can specify the color of the boat. Because they are so similar to rowboats, we can utilize the concept of inheritance to rapidly develop their class.

To begin with, they will inherit all of a rowboat's attributes as indicated by the arrow that connects them to the class `RowBoat` in Figure 8.1. This will save the time and cost associated with declaring a sailboat's five data members, writing and verifying the associated `set` and `get` methods, and writing and verifying the methods to draw them and to calculate their price.

To implement the expanding capability into the `SailBoat` class, the ability to specify the color of the boat when it is purchased, a four-parameter constructor is included in the `SailBoat` class. When a sailboat is constructed, the color of the new boat will be passed to the constructor's fourth parameter.

Figure 8.6 presents the code for the class `SailBoat`. Because we are using inheritance by including the `extends` clause at the end of its heading (line 3), the class consists of only ten lines of code. A sailboat's location, price per foot, length, and color can be stored in its inherited data members. In addition, the values stored in the data members can be fetched and set, the sailboat's price can be calculated, and the boat can be drawn on the game board using the methods inherited from the `RowBoat` class.

Invoking A Parent Class Constructor

Parent class constructors are *not* inherited, but they can be invoked within the code of a child class's constructor. The `SailBoat` class's constructor (lines 5–9 of Figure 8.6) invokes the parent class's constructor on line 7 and passes it the (x, y) location of the sailboat and its length. The invocation begins with the keyword **`super`**, rather than the name of the parent (also known as super) class, followed by an argument list. This is the syntax a child class uses to invoke a parent class constructor. Lines 14–16 of Figure 8.3 then execute and place the values passed to it into the `SailBoat` object's inherited data members `x`, `y`, and `length`.

NOTE *When a child class constructor invokes a parent class constructor, the invocation statement must be coded as the first line of the child class constructor.*

Every time a child class constructor is executed, a parent constructor must be executed before the remainder of the child class's constructor is executed. If an explicit invocation, such as the one on line 7 of Figure 8.6, is not included as the first line of a child class constructor, the parent's no-parameter constructor is automatically executed. In this case, the parent class must:

1. contain the code of a no-parameter constructor
2. contain no constructors at all, in which case the Java default constructor will be executed

If one of these two conditions is not met, and the first line of the child class is not an explicit invocation of a parent constructor, the child class will not translate.

Invoking a Parent Class Method

A child class method can invoke any parent method whose access is designated public or whose access is designated protected. We will discuss protected access, its use, and its implications later in this chapter.

Because the RowBoat class does not contain a four-parameter constructor, line 8 of Figure 8.6 invokes the method setColor to store the color of the new sailboat that was passed to the constructor on line 5. If the SailBoat class contained a setColor method, it would have executed. Because it does not, we move up the inheritance chain to the parent class. The parent class does contain a public setColor method (lines 61–64 of Figure 8.3), and the invocation on line 8 causes it to execute.

```
1   import java.awt.*;
2
3   public class SailBoat extends RowBoat
4   {
5     public SailBoat(int x, int y, int length, Color color)
6     {
7       super(x, y, length); //invoke parent constructor
8       setColor(color);   //access parent's protected data method
9     }
10  }
```

Figure 8.6
The class **SailBoat**.

An alternate and equivalent coding for line 8 of Figure 8.6 would be:

```
this.setColor(color);
```

Because all of the RowBoat class's set methods are public, line 7 of Figure 8.6 could have been coded as the following three lines. The one-line alternative shown in the figure is preferred.

```
setX(x); //or this.setX(x);
setY(y); //or this.setY(y);
setLength(length); //or this.setLength(length);
```

NOTE *The syntax used to invoke a parent method from within a child method is the same syntax (that was discussed in Chapter 7) used to invoke another method in its class.*

The application InheritanceBasics presented in Figure 8.7 creates two rowboats and a cyan-colored sailboat and then displays them on the left and right sides of the game board, respectively. The program's output is shown in Figure 8.8

The show method is invoked on line 20 of Figure 8.7 to display the sailboat whose location, size, and color is specified on line 9. Because a SailBoat object (sb1) invoked the method, the search for the method begins in the SailBoat class just as it does for classes that do not extend other classes. If the SailBoat class contained a show method with a graphics parameter, it would have been executed. Because it does not, we move up the inheritance chain to the parent class RowBoat. It contains a public show method with a graphics parameter (line 22 of Figure 8.3), and the invocation on line 20 causes it to execute. The inherited show method draws the sailboat object sb1.

Line 26 of the inherited show method invokes the method calculatePrice. The search for this method also begins in the SailBoat class because the show method that issued the invocation is operating on an instance of this class, `sb1`. Because the `SailBoat` class does not contain a `calculatePrice` method, the search continues up the inheritance chain, and `RowBoat` class's version of the method executes.

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class InheritanceBasics extends DrawableAdapter
{ static InheritanceBasics ge = new InheritanceBasics();
  static GameBoard gb = new GameBoard(ge, "Inheritance Basics");
  static RowBoat rb1 = new RowBoat(30, 150, 200);
  static RowBoat rb2 = new RowBoat(30, 250, 150);
  static SailBoat sb1 = new SailBoat(260, 150, 200, Color.CYAN);

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    rb1.show(g);
    rb2.show(g);
    sb1.show(g);
  }
}
```

Figure 8.7
The application **`InheritanceBasics`**.

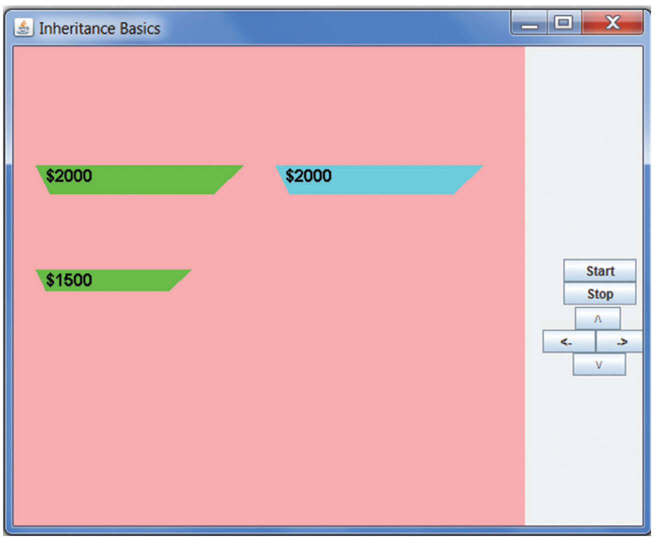


Figure 8.8
The output produced by the application **`InheritanceBasics`**.

8.3.2 Overriding Methods

Consistent with the concept of inheritance, the `SailBoat` class's four-parameter constructor added to the attributes inherited from its parent class. As previously mentioned, child classes can also modify the inherited attributes. The mechanism for modifying an inherited method is called *overriding* a method, which allows a child class to contain a method whose signature is exactly the *same* (same returned type, name, and parameter list) as the parent's method it is modifying.

Suppose that a second version of a sailboat, named `SailBoatV2`, was being offered for sale. This type of sailboat is not only delivered in a specified color, but it also has a mast installed on it. In order to properly display the new type of sailboat (with a mast), the `RowBoat`'s `show` method would be overridden. The modified version of the method, coded in the new `SailboatV2` class, would incorporate the drawing of the mast into its version of the method.

Because the UML diagram of the `RowBoat` shown in Figure 8.1 contains many more data members and methods than the UML diagram of the `SailBoat` class, we may be tempted to make the new class a child of the `RowBoat` class. If we do this, in addition to overriding the inherited `show` method, we would also have to rewrite the code of the `SailBoat` class's four-parameter constructor when the new class is coded. A better approach would be to take advantage of the fact that Java supports chain inheritance, which is depicted on the left side of Figure 8.2, and make the new class a child of the `SailBoat` class. The new class would then inherit all of the attributes (data members and methods) of both the `RowBoat` and the `SailBoat` classes, which gives it the ability to invoke the `SailBoat` class's four-parameter constructor. This approach to the design of the new class is depicted in Figure 8.9.

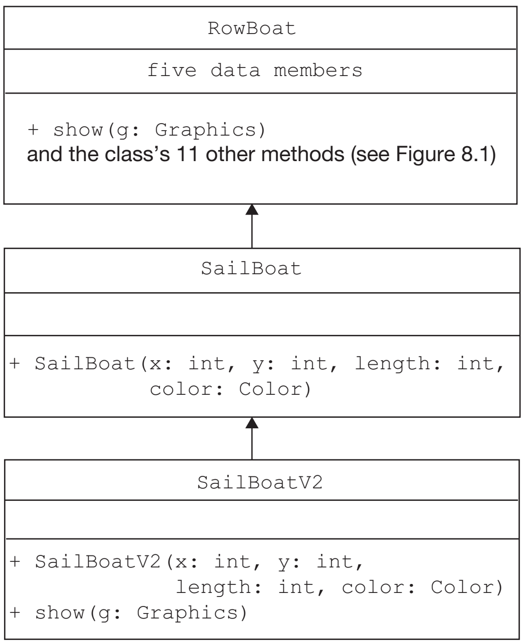


Figure 8.9
The inheritance chain of the class **`SailBoatV2`**.

The implementation of the class `SailBoatV2` is shown in Figure 8.10. As specified in Figure 8.9, it extends the class `SailBoat` (line 3). Its four-parameter constructor (lines 5–8) simply invokes its parent's four-parameter constructor on line 7, passing it the (x, y) location, length, and color of the new sailboat that was passed to it. It also overwrites the `show` method it inherits from the `RowBoat` class (lines 11–17).

```
1   import java.awt.*;
2
3   public class SailBoatV2 extends SailBoat //overriding a parent method
4   {
5     public SailBoatV2(int x, int y, int length, Color color)
6     {
7       super(x, y, length, color); //invoke the parent’s constructor
8     }
9
10    @Override //translator verified an inherited method has this signature
11    public void show(Graphics g) // overwrites the parent's method
12    {
13      super.show(g); //invoke the parent’s method to draw the boat
14      g.setColor(Color.BLACK); //draw the mast
15      g.fillRect(getX() + getLength()/2, getY() - getLength()/2,
16                 3, getLength()/2);
17    }
18  }
```

Figure 8.10
The class **`SailBoatV2`**.

Invoking a Parent's Version of an Overwritten Method

The overridden version of the `show` method coded in the `SailBoatV2` class begins by invoking its inherited `show` method (line 13 of Figure 8.10) to draw the hull of the sailboat. As when invoking an inherited constructor, the keyword **`super`** is used in the invocation. When invoking a nonconstructor inherited method, the keyword **`super`** is followed by a dot.

The `SailBoatV2` class's inheritance chain is used to locate the invoked method. The syntax **`super.`** that proceeds the name and argument list of the invocation on line 13 tells the translator that we do not want the search to begin for the invoked method in the `SailBoatV2` class but rather to begin the search in its parent class. Otherwise, the `SailBoatV2` class's `show` method would invoke itself.

Because the `RowBoat` does contain a `show` method whose parameter is a `Graphics` object (line 22 of Figure 8.3), this method executes and draws the hull of the boat. Then, lines 14–16 of Figure 8.10 incorporate the modification to this method by drawing the sailboat's mast. If the search up the inheritance chain did not locate a `show` method whose parameter was a `Graphics` object, the translation of the `SailBoatV2` class would have ended in a translation error.

The @Override directive

The `@Override` directive that appears on line 10 of Figure 8.10 instructs the translator to search up the `SailBoatV2` class's inheritance chain for a method with the same signature that is coded on the line that follows it. If it cannot find a method with that signature, the translation ends in an error.

It is good programming practice to include this translation directive before the signature of a method that is meant to override an inherited method. Suppose the method name or parameter list typed on line 11 was syntactically correct but did not match the parameter list of the method it was overriding. For example, suppose the signature with the method's name was misspelled as shown below:

```
11  public void shown(Graphics g) //does NOT override inherited show method
```

Without the directive on line 10 of Figure 8.10, the class would translate, but the inherited `show` method would not have been overridden. The client invocation on line 20 of Figure 8.11 would then cause the inherited method to execute, and the sailboat would be drawn without a mast.

Figure 8.11 presents the application `OverridingMethods`, which is the same application presented in Figure 8.7, except the sailboat it creates on line 9 is now a `SailBoatV2` object. As a result, the `SailBoatV2` class's overridden `show` method is invoked on line 20, and the sailboat is drawn with a mast. The output it produces is shown in Figure 8.12

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3
4    public class OverridingMethods extends DrawableAdapter
5    { static OverridingMehtods ge = new OverridingMethods();
6      static GameBoard gb = new GameBoard(ge, "OVERRIDING METHODS");
7      static RowBoat rb1 = new RowBoat(30, 150, 200);
8      static RowBoat rb2 = new RowBoat(30, 250, 150);
9      static SailBoatV2 sb1 = new SailBoatV2(260, 150, 200, Color.CYAN);
10
11     public static void main(String[] args)
12     {
13       showGameBoard(gb);
14     }
15
16     public void draw(Graphics g)
17     {
18       rb1.show(g);
19       rb2.show(g);
20       sb1.show(g);
21     }
22   }
```

Figure 8.11
The application **`OverRidingMethods`**.

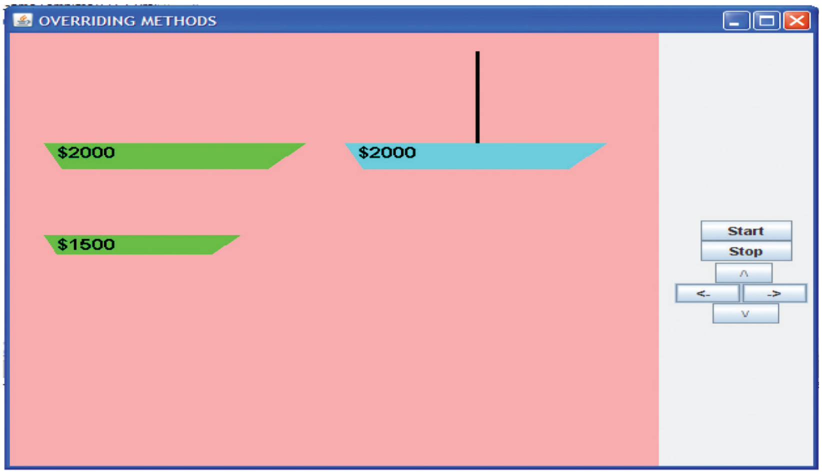


Figure 8.12
The output produced by the application **`OverRidingMethods`**.

Final Methods

A method can be declared to be a `final` method by coding the keyword **`final`** in its signature immediately after the method's access modifier. When a method is declared to be final, it cannot be overridden by a child class. An attempt to do so results in a translation error. For example, if the signature of the `show` method on line 22 of the `RowBoat` class (Figure 8.3) was coded as shown below, the `SailBoatV2` class shown in Figure 8.10 could not override it.

```
22      public final void show(Graphics g)
```

Methods that enforce security on systems are usually declared to be final to prevent them from being overridden.

Overriding versus Overloading Methods

The concepts of overriding and overloading methods are often confused because both concepts can be used to code a new method that has the same name as an existing method. In addition, the two topics are often considered to be more restrictive than they actually are. Before concluding our discussion of overriding methods, we will discuss the differences between, and commonalities shared by, these concepts.

One difference between the concepts of overriding and overloading methods also allows us to identify which concept is being used. When two methods in an inheritance chain have the same name and the same parameter list, the concept of *overriding* methods is being used. When two methods in an inheritance chain, or within the same class, have the same name and different parameter lists (i.e., either the number and/or type of the parameters are different), the concept of *overloading* methods is being used.

A second difference is that the concept of overriding a method cannot be used to modify the functionality of a method coded in its class because two methods with the same name and parameter list cannot be coded in the same class. The code of an overridden method and the code of the method that overrides it must appear in two different classes in an inheritance chain. The concept of overloading a method is less restrictive. The code of an overloaded method and that of the method it overloads can appear in the same class or in different classes within an inheritance chain.

The concepts of overriding and overloading methods have many things in common:

- both concepts are used to modify or expand the functionality of an existing method
- both concepts can be used to produce a new method that has the same name as an existing method
- both static and nonstatic methods can be overridden and overloaded
- a child class can overload and override any of its inherited methods
- the translator always uses the same technique to locate and identify a method that is being invoked regardless of the whether the method has been overridden or overloaded

This section will conclude with a summary of the search path the translator uses to locate an invoked overridden or overloaded method, and a summary of the syntax used to invoke inherited methods.

Summary of the Method Search Path

When an overloaded or overridden method is invoked, the translator uses the same search process to locate the method that it uses to locate all invoked methods. The class of the object's reference variable, or the class of an invoked static method, is searched for a method whose name and parameter list matches the name and argument list in the invocation statement. If a match is not found in that class, the search continues up the class's inheritance chain.

Summary of the Inherited Method Invocation Syntax

As we have learned, a child class method can invoke a method inherited from its parent. If the method is not overridden in the child class, the method is invoked by coding the name of the method followed by an argument list. When the method is a static method, this syntax is preceded by the name of the parent class followed by a dot. If the parent method is overridden and is not a static method, the invocation is preceded by the keyword `super` followed by a dot.

8.3.3 Extending Inherited Data Members

A child class's ability to extend the attributes it inherits is not limited to overriding and overloading inherited methods or including new methods in its class. It can also extend the data members it inherits by declaring additional data members inside its class definition. Figure 8.13 presents the inheritance chain of a class named `SailBoatV3`, which is a sailboat delivered with a mast and a sail. To facilitate the implementation of the new class, it extends the `SailboatV2` class because, in all other aspects, this new type of sailboat is a `SailBoatV2` object.

As shown in its UML diagram, the `SailBoatV3` class will override its inherited `calculate-Price` method. The new version of this method will use the class's new data members `sailArea` and `pricePerSquareFoot` to calculate and include the cost of the sail in the boat's price. The boat's `sailArea` will be specified using the last parameter of the class's five-parameter constructor. In addition, the inherited `show` method will be overwritten. This version of the method will be used to draw the sail on the boat's mast.

NOTE *The name of a data member added to a child class can be the same as the name of an inherited data member, but it is good programming practice to avoid duplicating inherited data member names.*

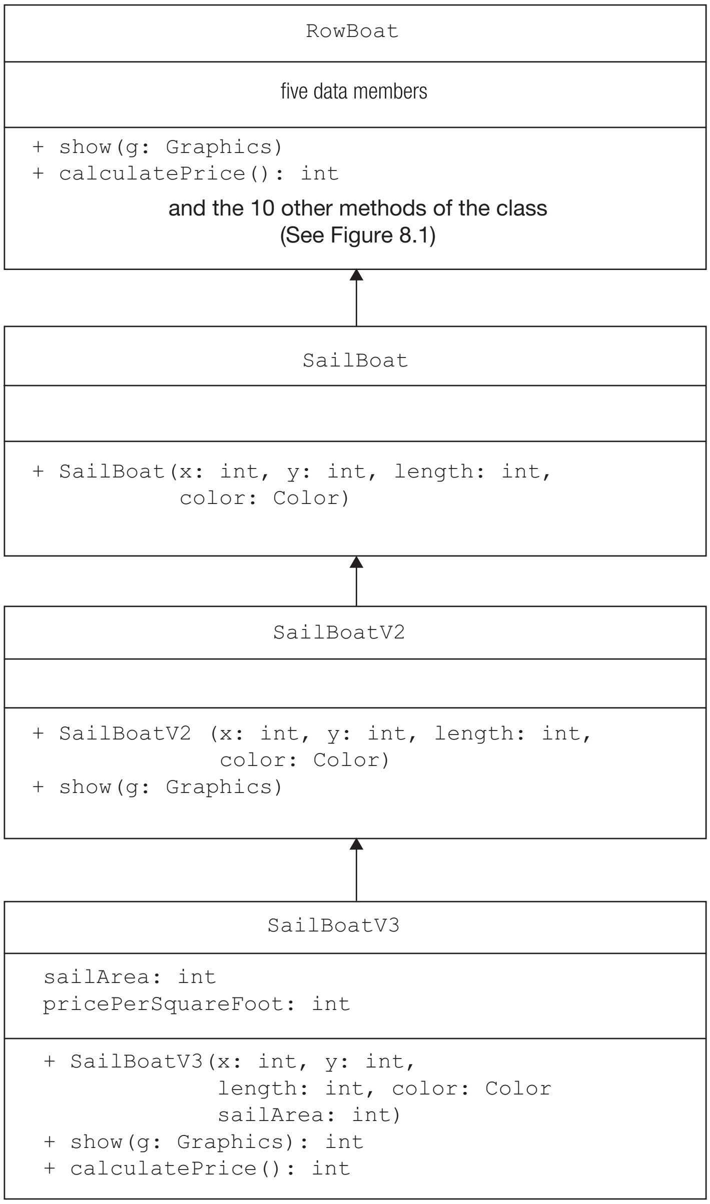


Figure 8.13
The inheritance chain of the class **`SailBoatV3`**.

Figure 8.14 shows the implementation of the `SailBoatV3` class that extends the class `SailBoatV2` (line 3). Its two additional data members, `pricePerSquareFoot` and `sailArea`, are declared on lines 5 and 6. Line 12 of the class's constructor stores the sail area passed to its last parameter in the data member `sailArea`. Line 19 of the expanded `calculatePrice` method multiplies these data members to determine the sail's price and then adds this product to the price returned from line 18's invocation of `RowBoat`'s version of the method.

Lines 25–31 implement additional functionality of the overridden `show` method by defining the (x, y) coordinates of the triangular sail's vertices and then drawing the sail. Line 32 invokes the class's inherited `show` method to draw the boat's hull and mast.

Figure 8.15 presents the application `ExtendingDataMembers`, which is the same application presented in Figure 8.11, except it creates a yellow instance (`sb2`) of the class `SailBoatV3` on line 10. As a result, the `SailBoatV3` class's version of the `show` method is invoked on line 22, and the

```
1    import java.awt.*;
2
3    public class SailBoatV3 extends SailBoatV2
4    {
5      private static int pricePerSquareFoot = 2;
6      private int sailArea; // additional data members
7
8      public SailBoatV3(int x, int y, int length,
9                        Color color, int sailArea)
10     {
11       super(x, y, length, color);
12       this.sailArea = sailArea;
13     }
14
15     @Override
16     public int calculatePrice() //invokes the method it overrides
17     {
18       int hullPrice = super.calculatePrice(); //invokes RowBoat's method
19       return hullPrice + sailArea * pricePerSquareFoot;
20     }
21
22     @Override
23     public void show(Graphics g)
24     {
25       int[] xSail = {getX() + getLength()/2, getX(),
26                      getX() + getLength()/2,};
27       int[] ySail = {getY() - getLength()/2, getY() - getLength()/8,
28                      getY() - getLength()/8};
29
30       g.setColor(Color.WHITE); //draw the sail
31       g.fillPolygon(xSail, ySail, xSail.length);
32       super.show(g);
33     }
34   }
```

Figure 8.14
The class **SailBoatV3**.

yellow sailboat is drawn with a sail as shown in the bottom right portion of Figure 8.12. Although the lengths of the two sailboats declared in the application (lines 9–10) are both 200, the cost of the second boat (`sb2`) is higher because of the $600 additional cost of the boat's 300-square-foot sail (at $2 per square foot as per line 5 of Figure 8.14).

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3
4    public class ExtendingDataMembers extends DrawableAdapter
5    { static ExtendingDataMembers ge = new ExtendingDataMembers ();
6      static GameBoard gb = new GameBoard(ge, "EXTENDING DATA MEMBERS");
7      static RowBoat rb1 = new RowBoat(30, 150, 200);
8      static RowBoat rb2 = new RowBoat(30, 250, 150);
9      static SailBoatV2 sb1 = new SailBoatV2(260, 150, 200, Color.CYAN);
10     static SailBoatV3 sb2 = new SailBoatV3(260, 300, 200, Color.YELLOW, 300);
11
12     public static void main(String[] args)
13     {
14       showGameBoard(gb);
15     }
16
17     public void draw(Graphics g)
18     {
19       rb1.show(g);
20       rb2.show(g);
21       sb1.show(g);
22       sb2.show(g);
23     }
24   }
```

Figure 8.15
The application **`ExtendingDataMembers`**.

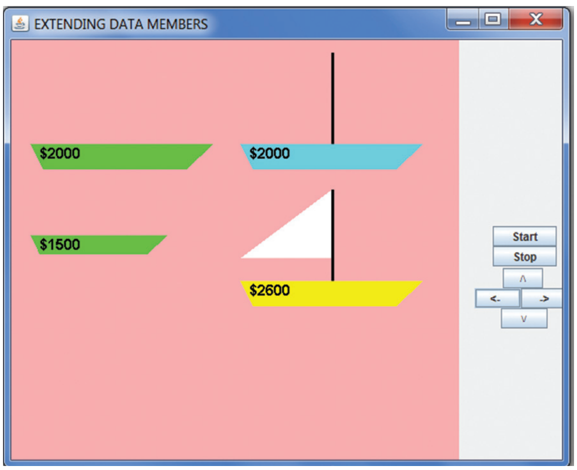


Figure 8.16
The output produced by the application **`ExtendingDataMembers`**.

Parent Class Methods Invoking Child Class Methods

As previously discussed at the end of Section 8.3.1, when the `RowBoat` class's `show` method invokes the method `calculatePrice` (line 26 of Figure 8.3), the search for the method begins in the class of the object that invoked the `show` method. When line 22 of Figure 8.15 executes, the search for the `calculatePrice` method begins in `sb2`'s class, `SailBoatV3`. Because this class overrides the inherited version of the `calculatePrice` method, `sb2`'s price is calculated by its version of the method coded on lines 16–20 of Figure 8.14. This is precisely what should happen because otherwise the cost of the sail would not be included in the price of the sailboat `sb2`. The Java process used to locate invoked methods, which is summarized at the end of Section 8.3.2, causes the `RowBoat` class's `show` method to invoke the `SailBoatV3` class's `calculatePrice` method to determine the price of a `SailBoatV3` sailboat instance.

8.4 USING INHERITANCE IN THE DESIGN PROCESS

The time and effort required to create a new class can be greatly reduced if we can extend the attributes of an existing class using the basic techniques of inheritance discussed in the previous section. These techniques include inheriting methods and data members into the new class, overriding and overloading these methods to change and extend their functionality, and adding new methods and data members to the new class. But even if there are no existing classes that provide some of the functionality of the classes specified for a new program, the time and cost to develop the program can still be reduced using the concept of inheritance during the design process.

Suppose your Uncle Ed asked you to develop a Java program to "keep track of the inventory" of a boat store he was about to open that will carry rowboats, sailboats, and powerboats. After several follow-up conversations with him, you have determined that "keeping track of the inventory" means knowing the location of each boat on his storage lot, knowing each boat's price and size, and other details that are particular to the type of the boat such as the number of oars, the sail area, and the horsepower of a powerboat. Translating all of this into an OPP design, you concluded there will be three worker classes in the program and produced the UML diagrams shown in Figure 8.17.

Before proceeding to the coding phase of Uncle Ed's (or any other) program, we should apply the basic concepts of inheritance previously discussed in this chapter and the other more advanced inheritance concepts, such as *abstract classes*, to the design process.

8.4.1 Abstract Classes

After the UML diagrams that describe the objects that will be part of the program are prepared, their data members and methods should be compared to determine their commonalities. An examination of the data members of the classes specified in Figure 8.17 reveals that the first five data members in all three classes are the same. In addition, the signatures of all of their methods, except for their constructors, are the same. If we were to give these UML diagrams to three programmers to implement, each programmer would have to code the same five data members into their class and write the same six `set` and `get` methods to change and fetch the values of the data members. In addition, the code of their `calculatePrice`, `show`, and `toString` methods would share some similar code.

```
RowBoatV2
----------------------------------------
- pricePerFoot: int
- x: int
- y: int
- length : int
- color: Color
- oars : int
----------------------------------------
+ RowBoatV2 (x: int , y: int,
             length:  int , c: Color,
             oars:  int)
+ calculatePrice(): int
+ show(g: Graphics)
+ toString(): String
+ getX(): int
+ getY(): int
+ getLength(): int
+ getColor(): Color
+ setX(x: int)
+ setY(y: int)
```

```
SailBoatV4
----------------------------------------
- pricePerFoot: int
- x: int
- y: int
- length : int
- color: Color
- sailArea: int
----------------------------------------
+ SailBoatV2 (x: int , y: int,
              length:  int , c: Color,
              sailArea : int)
+ calculatePrice(): int
+ show(g: Graphics)
+ toString(): String
+ getX(): int
+ getY(): int
+ getLength(): int
+ getColor(): Color
+ setX(x: int)
+ setY(y: int)
```

```
PowerBoat
----------------------------------------
- pricePerFoot: int
- x: int
- y: int
- length : int
- color: Color
- horsePower: int
----------------------------------------
+ PowerBoat (x: int , y: int,
             length:  int , c: Color,
             horsepower:   int)
+ calculatePrice(): int
+ show(g: Graphics)
+ toString(): String
+ getX(): int
+ getY(): int
+ getLength(): int
+ getColor(): Color
+ setX(x: int)
+ setY(y: int)
```

Figure 8.17
The class design of a boat store's inventory application.

To avoid this duplication among the three classes, a forth class named `Boat` is added to the design as shown at the top of Figure 8.18. This class will be an *abstract class*. A class is designated to be abstract by including the keyword **`abstract`** in its heading. For example:

```
public abstract class Boat //an abstract class
```

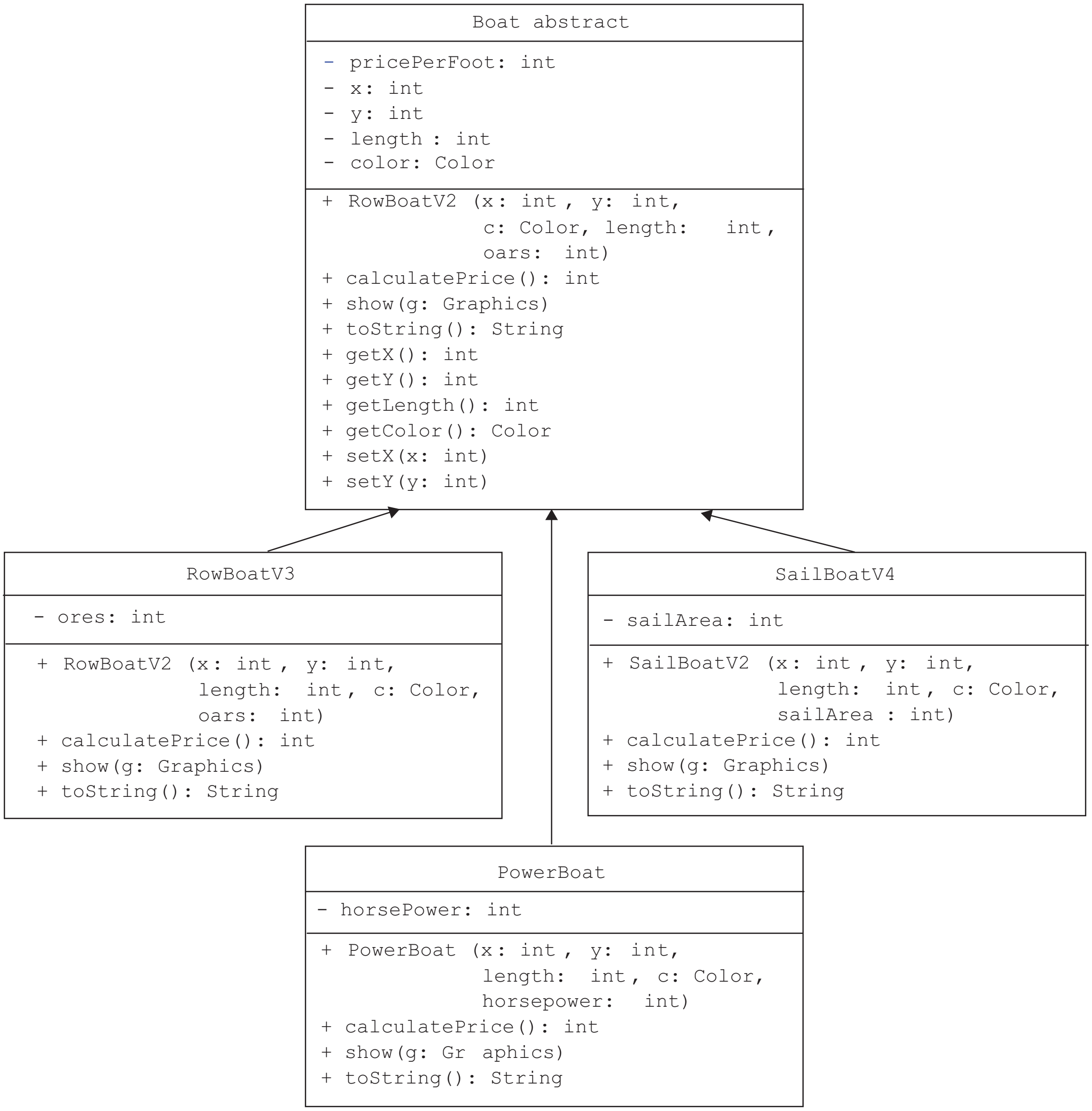


Figure 8.18
The use of inheritance in the design of a boat store's inventory application.

An abstract class is used to collect data members and methods that are common to several classes into one class. It is not meant to be the class of one of the types of objects that will make up a program. For example, you cannot purchase an instance of a `Boat` object from Uncle Ed. He only sells rowboats, sailboats, and powerboats. Consistent with this use of abstract classes, an attempt to declare an object in an abstract class results in a translation error.

As shown in the bottom portion of Figure 8.17, after the abstract `Boat` class is added to the design of Uncle Ed's program, the duplicated five data members and the `set` and `get` methods are eliminated from the original three classes and moved into the UML diagram of the `Boat` class.

Then, as indicated by the arrows in the figure, the original three classes become subclasses of the `Boat` class. By simply including the keyword **`extends`** in the heading of the classes they code, the authors of the original three classes no longer have to code five of their class's six data members or the six `set` and `get` methods. But the reduction in effort does not end here.

Each of the original classes also contained methods to draw the boat it defines and calculate its price as well as a `toString` method. The signatures of these methods are the same in all three classes, and your conversations with Uncle Ed revealed that they share some common functionality. He has told you that the base price of each type of boat is calculated in the same way because they share a common hull. In reflecting on what he said, you realize that this introduces some common functionality into the `show` methods because each boat's hull will look the same. It also introduces some common functionality into the `calculatePrice` method because each boat's hull will be priced in the same way. Because each of the `toString` methods will return the annotated values of the classes' first five data members, these methods also share some common functionality.

Methods in the child classes that have the same signature and share some common functionality also become part of the parent class's UML diagram, as shown in the UML diagram of the `Boat` class, which now includes a `calculatePrice`, a `show`, and a `toString` method. Unlike the `set` and `get` methods that were eliminated from the child classes, some trace of these relocated methods are retained in the UML diagrams of the child classes to provide the functionality that is not common to each child class.

For example, the functionality of calculating the additional cost of a powerboat's motor would have to be retained in the `PowerBoatV2` class, and the cost of a rowboat's oars would be calculated in the `RowBoat` class. In the design presented in Figure 8.18, the child classes provide additional functionality by overriding the `Boat` class's `calculatePrice`, `show`, and `toString` methods. The code of these methods would invoke `Boat`'s version of the method to calculate and return the price of the boat's hull, draw the boat's hull, and to build and return a string containing the annotated versions of `Boat`'s five data members.

Comparing the UML diagrams in Figures 8.17 and 8.18, we can see that the use of inheritance in the design presented in Figure 8.18 has significantly reduced the effort required to produce Uncle Ed's program. The number of data members to be coded by the programmers has been reduced from 18 to 8, and the number of methods has been reduced from 30 to 22. In addition, the constructor in the `Boat` class will implement the commonality within the constructors in the other three classes, which will reduce the effort required to produce the constructors in the other three classes.

Figures 8.19–8.22 present the implementation of the design depicted in Figure 8.18. The `Boat` class is declared abstract on line 4 of Figure 19. Consistent with our design philosophy, it includes all of the data members common to the other three classes, and its methods provide the functionality shared by the other three classes. Those classes extend the `Boat` class (line 3 of Figures 8.20–8.22). They each include a data member particular to their type of boat (e.g., line 5 of Figure 8.20), and their methods invoke `Boat`'s methods (e.g., lines 9, 15, 21, and 31 of Figure 8.20) before adding the functionality particular to their type of boat. For example, line 16 of Figure 8.20 computes the cost of the oars, lines 22-26 draw the oars, and line 31 adds the number of oars to `toStrings` returned strings. The `implements` clause in the heading of the class `Boat` (line 4 of Figure 8.19) will be discussed in Section 8.7.

The application `DesignTechniques` shown in Figure 8.23 uses the new design to add a rowboat with four oars (line 8), a sailboat with a 200-square-foot sail (line 9), and a powerboat with a 400-horsepower motor (line 10) to Uncle Ed's inventory. The boats are then displayed on his lot (Figure 8.24).

```
import java.awt.*;
import java.io.Serializable;

public abstract class Boat implements Serializable //contains attributes
{                                                  //common to all boats
  private static int PRICE_PER_FOOT = 10;
  private int x, y, length; //data members common to all types of boats
  private Color color;

  public Boat(int x, int y, int length, Color color)
  {
    this.x = x;
    this.y = y;
    this.length = length;
    this.color = color;
  }
  public int calculatePrice() //will be overridden
  {
    return length * PRICE_PER_FOOT;
  }
  public void show(Graphics g) //will be overridden
  {
    int[] xBoat = {getX() , getX() + length, getX() + 6*length/7,
                   getX() + length/14};
    int[] yBoat = {getY(), getY(), getY() + length/7,
                   getY() + length/7};
    int price = calculatePrice();
    g.setColor(color);
    g.fillPolygon(xBoat, yBoat, xBoat.length);
    g.setColor(Color.BLACK);
    g.setFont(new Font("Arial", Font.BOLD, 16));
    g.drawString("$" + String.valueOf(price), x + 10, y + 16);
  }
  public String toString() //will be overridden
  {
    return "Location: (" + x + ", " + y +"), length: " + length +
                       ",Color: " + color;
  }
  public int getX() //get & set methods common to all types of boats
  {
    return x;
```

```
  }
  public int getY()
  {
    return y;
  }
  public int getLength()
  {
    return length;
  }
  public Color getColor()
  {
    return color;
  }
  public void setX(int x)
  {
    this.x = x;
  }
  public void setY(int y)
  {
    this.y = y;
  }
}
```

Figure 8.19
The abstract class **Boat**.

```
import java.awt.*;

public class RowBoatV2 extends Boat
{
  private int oars; //extended (additional) data member

  public RowBoatV2(int x, int y, int length, Color c, int oars)
  {
    super(x, y, length, c);
    this.oars = oars;
  }
  @Override
  public int calculatePrice() //overrides parent method
  {
    int hullPrice = super.calculatePrice();
    return hullPrice + oars * 10;
  }
  @Override
  public void show(Graphics g) //overrides parent method
  {
    super.show(g);
```

```
    g.setColor(Color.BLACK);
    for(int i = 1; i <= oars; i++) //each ore
    {
      g.fillRect(getX() + i*10, getY() - 20, 2, 20); //handle
      g.fillOval(getX() + i*10-2, getY() - 30, 6, 10); //paddle
    }
  }
  public String toString() //overrides parent method
  {
    return super.toString() + ", Oars: " + oars;
  }
}
```

Figure 8.20
The child class **RowBoatV2**.

```
import java.awt.*;

public class SailBoatV4 extends Boat
{
  private int sailArea; //extended (additional) data member

  public SailBoatV4(int x, int y, int length,
                   Color color, int sailArea)
  {
    super(x, y, length, color);
    this.sailArea = sailArea;
  }
  @Override
  public int calculatePrice() //overrides parent method
  {
    int hullPrice = super.calculatePrice();
    return hullPrice + sailArea * 2;
  }
  @Override
  public void show(Graphics g) //overrides parent method
  {
    int[] xSail = {getX() + getLength()/2, getX(),
                   getX() + getLength()/2};
    int[] ySail = {getY() - getLength()/2, getY() - getLength()/8,
                   getY() - getLength()/8};

    super.show(g);
    g.setColor(Color.BLACK); //draw the mast
    g.fillRect(getX() + getLength()/2, getY() - getLength()/2, 3,
               getLength()/2);
    g.setColor(Color.WHITE); //draw the sail
    g.fillPolygon(xSail, ySail, xSail.length);
  }
```

```
  public String toString() //overrides parent method
  {
    return super.toString() + ", Sail Area: " + sailArea;
  }
}
```

Figure 8.21
The child class **SailBoatV4**.

```
import java.awt.*;

public class PowerBoat extends Boat
{
  private int horsePower; //extended (additional) data member

  public PowerBoat(int x, int y, int length,
                   Color color, int horsePower)
  {
    super(x, y, length, color);
    this.horsePower = horsePower;
  }
  @Override
  public int calculatePrice() //overrides parent method
  {
    int hullPrice = super.calculatePrice();
    return hullPrice + horsePower * 3;
  }
  @Override
  public void show(Graphics g) //overrides parent method
  {
    int[] xSail = {getX() + getLength()/2, getX(),
                   getX() + getLength()/2,};
    int[] ySail = {getY() - getLength()/2, getY() - getLength()/8,
                   getY() - getLength()/8};

    super.show(g);
    g.setColor(Color.BLACK); //draw the shaft
    g.fillOval(getX() - 13, getY() + getLength()/7 , 30, 4);
    g.setColor(Color.GRAY); //draw the propeller
    g.fillOval(getX() - 20, getY() + getLength()/7, 20, 6);
    g.fillOval(getX() - 13, getY() + getLength()/7 - 7, 6, 20);
  }
  public String toString() //overrides parent method
  {
    return super.toString() + ", Horsepower: " + horsePower;
  }
}
```

Figure 8.22
The child class **PowerBoat**.

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class DesignTechniques extends DrawableAdapter
{
  static DesignTechniques ge = new DesignTechniques();
  static GameBoard gb = new GameBoard(ge, "Design Techniques");
  static RowBoatV2 rb1 = new RowBoatV2(50, 200, 120, Color.YELLOW, 4);
  static SailBoatV4 sb1 = new SailBoatV4(220, 200, 200, Color.GREEN, 200);
  static PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);

  public static void main(String[] args)
  {
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    rb1.show(g);
    sb1.show(g);
    pb1.show(g);
  }
}
```

Figure 8.23
The application **DesignTechniques**.

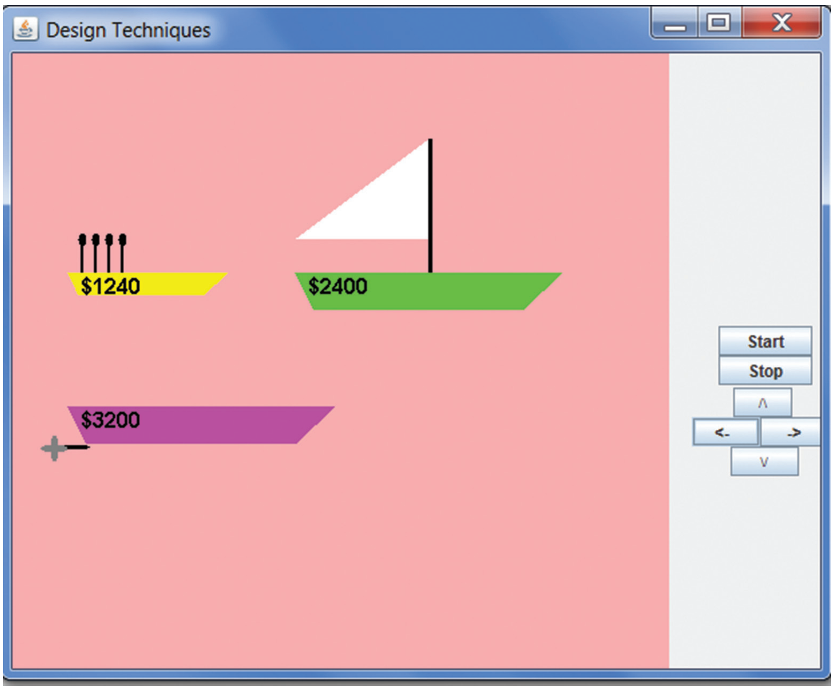


Figure 8.24
The output produced by the application **DesignTechniques**.

8.4.2 Designing Parent Methods to Invoke Child Methods

In addition to creating an abstract class to collect the common attributes shared by the various objects that will be part of a program, there are other inheritance concepts that should be considered when designing and coding the methods in an abstract or nonabstract parent class. One of them is the ability of a parent method to invoke a child class's method.

As discussed at the end of Section 8.3.3, the code of a method in a super class that is operating on an instance of a direct or indirect subclass can invoke a method coded in the subclass. The syntax used is the familiar syntax of coding the method's name followed by the appropriate argument list. For example, if a nonstatic method named `extras` is added to the child class `RowBoatV2`, shown in Figure 8.20, then it could be invoked inside any of the methods of its parent class `Boat`, shown in Figure 8.19, by coding: `extras();`

This presents an alternate approach to adding functionality to an inherited method during the design phase. Instead of child classes overriding an inherited method, they simply include a method that implements their version of the added functionality, and the code of the parent's inherited method includes an invocation of this method. The signature of the method added to the child classes must be the same in all of the child classes that intend to add functionality to the inherited method.

If this design approach was taken to calculate the price of a rowboat, sailboat, and powerboat, the `Boat` class's `calculatePrice` method, lines 17–20 of Figure 8.19, would be changed to the version of the method at the top Figure 8.25. This version of the method invokes the method `extras` to calculate an additional cost to be added to the price of the hull for oars, a sail, or a motor. The empty version of the method extras shown at the bottom of the figure has to be added to the `Boat` class or it will not compile. The child classes' `calculatePrice` methods in Figures 8.20, 8.21, and 8.22 would be replaced with the code of the `extras` method shown in the upper, middle, and lower portions of Figure 8.26, respectively.

```
public int calculatePrice()
{
  return length * PRICE_PER_FOOT + extras();
}

public int extras()
{
  return 0;
}
```

Figure 8.25
A parent method that invokes the child's method **`extras`** and the empty implementation of the **`extras`** method coded in the parent class.

```
public int extras() //RowBoatV2's version of the method extras
{
   return oars * 10;
}

public int extras() //SailBoatV4's version of the method extras
{
   return sailArea * 2;
}

public int extras() //PowerBoat's version of the method extras
{
   return horsePower * 3;
}
```

Figure 8.26
Three child class implementations of the method **`extras`**

8.4.3 Abstract Parent Methods

An alternative to including the empty version of the method `extras` shown at the bottom of Figure 8.25 is to code it as an abstract method in the parent class, in this case, the class `Boat` (Figure 8.19). Abstract methods include the keyword **`abstract`** in their signature before their returned type. In addition, their signature ends with a semicolon. As shown below, they do not contain an open and close brace or any code.

```
public abstract int extra();
```

When this approach is used, the parent class must be declared abstract because the parent class no longer contains an implementation of a method it invokes. In addition, the translator will verify that each nonabstract class that inherits directly or indirectly from the abstract class implements a method whose signature matches the signature of the abstract method. If it does not, the child class will not translate. In effect, the inclusion of an abstract method in a parent class is a promissory note, enforced by the translator, that child classes will implement (override) the abstract method.

The use of abstract methods in a super class is considered to be good programming practice when:

- The super class will not be instantiated. Its sole purpose is to collect the common data members and functionality shared by its direct and indirect subclasses.
- Most of its direct and indirect subclasses will add functionally to the super class method that invokes the abstract method.

In the event that a subclass does not need to add functionality to the super class method that invokes an abstract method, the subclass would implement the abstract method with an empty code block.

8.4.4 Final Classes

A class can be declared to be a **`final`** class by coding the keyword **`final`** in its heading immediately after the class's access modifier. When a class is declared **`final`**, it cannot be extended. An attempt to do so results in a translation error. For example, if the heading of the `PowerBoat` class (line 3 of Figure 8.22) was coded as shown below, then it could not be a parent class.

```
3    public final class PowerBoat extends Boat
```

Classes that contain methods that enforce security on systems are usually declared to be **`final`** to prevent their methods from being overridden.

A class cannot be declared **`abstract`** and **`final`** because it would be rendered useless. If it were abstract, instances of the class could not be created; if it were also final, it could not be extended. In short, nothing could be done with it.

8.4.5 Protected Data Members

When a method or a data member in a class is designated to have private access, the method can only be invoked and the data member can only be directly accessed by the methods defined inside the class. A private method cannot be invoked by methods defined outside of its class. A private data member can be indirectly accessed by a method defined in another class by invoking its `set` and `get` methods. When a method or a data member is designated to have public access the method can be invoked, and the data member can be directly accessed, by methods defined outside of its class.

A class's methods and data members can also be declared to have protected access by beginning their declaration with the keyword **`protected`**.

```
protected int count;
```

The access modifier **`protected`** is less restrictive than **`private`** access and more restrictive than **`public`** access. Restrictions imposed by protected access are package dependent.

When a method or a data member is designated to have **`protected`** access, the method can be invoked, and the data member can be directly accessed, by methods defined inside its class *and* its direct and indirect subclasses. As illustrated in the bottom and top left portions of Figure 8.27, the access is permitted whether or not the parent's protected data members/methods and the child classes are defined in the same package.

Methods in nonchild classes can only access another class's protected methods, and directly access its protected data members, if the nonchild class is defined in the same package as the protected methods and data. This is illustrated in the top right and bottom portions of Figure 8.27.

When an access modifier (**`public`**, **`private`**, or **`protected`**) is not specified for a class's data member or method, it can be directly accessed by the code of any other class in the package. This is referred to as *package* access.

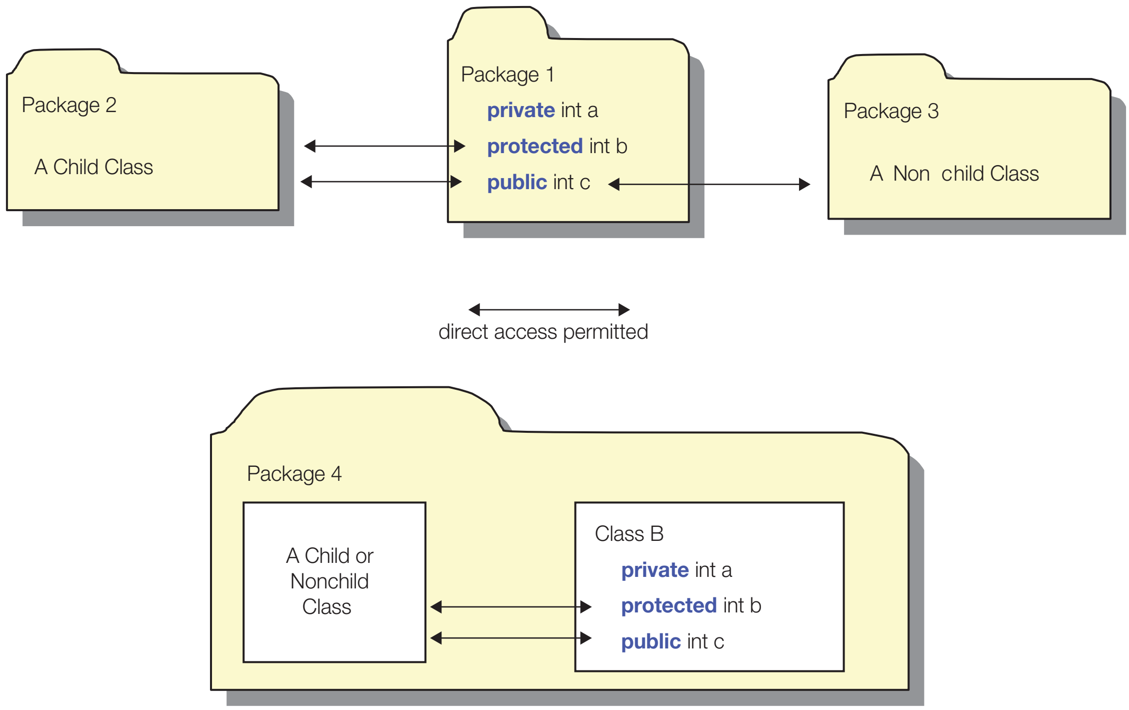


Figure 8.27
Restrictions imposed by `private, protected`, and `public access`.

NOTE *Protected methods and data members are hidden from nonchild classes defined in separate packages.*

8.4.6 Making a Class Inheritance Ready: Best Practices

When designing a class, we should always consider the possibility that other classes may extend it. If for some reason we want to prevent the class from being extended, the class is declared `final`. If a nonfinal class contains a constructor, it should also contain a no-parameter constructor even if its code block is empty, otherwise the child class's constructors will always have to explicitly invoke a parent constructor. A parent class that was created to collect the methods and data members common to other classes should be declared abstract.

Each method in a nonfinal class should be examined to decide if its functionality could be compromised if it were overridden in a subclass. When that is the case, the method should be declared `final`. To permit restricted modifications to a `final` method, the method should invoke an abstract method whose signature is defined within the `final` method's class, then subclasses can add functionality to a `final` method by implementing the abstract method. This abstract method approach should also be used when the algorithm of a parent class method requires that child classes add functionality particular to them because the translator will verify that the child classes implement the abstract method. This approach should also be considered when it is anticipated that a child class is likely to override a parent method.

A parent data member that stores a constant should always be declared as `final` to prevent it from being changed. Generally speaking, it is good programming practice to declare all nonfinal data members in a parent class to be private rather than protected and, where appropriate, include `set` and `get` methods to permit access to them. This maintains the encapsulation of the data members, which eliminates the possibility that a method in a child class could unintentionally access the data members by using a variable with the same name that it neglected to declare.

8.5 POLYMORPHISM

Polymorphism is the idea that something can exist in several different forms. We have already discussed several uses of polymorphism in computer science. One use of polymorphism we discussed is overloading methods. A method can be overloaded or morphed into a new form, with the new form having a different parameter list. These different forms of the method can be coded inside the same class, such as constructors with several different parameter lists, or one form of a method's parameter list can be coded in a super class, and another form could be coded in its subclass.

Other uses of polymorphism occur within the concept of inheritance. Two of these uses were already discussed. The first is the inheritance concept of overriding methods. In this use of polymorphism, one form of a method exists in the super class, and another form of the method, with the exact same signature, exists in the subclass. The second involves the invocation of a method by a parent method. Due to the search path used to locate the invoked method at run time, this invocation could take on the form of an invocation of a parent class method or a child class method with the same signature.

Polymorphism is also used to indicate that a child can take on the form of a parent because a child is also a parent. This should not be surprising because we have already learned that a child class inherits everything from a parent: all of its data members and all of its methods (except for its constructors). This polymorphic inheritance concept opens up a set of programming practices that makes our programs easier to write and easier to understand. In the remainder of this section, we will discuss these programming practices and the syntax used to incorporate them into the programs we write.

8.5.1 Parent and Child References

Because a child object is also considered to be a parent object, a parent reference variable can store the address of, or point to, a child object. This is a form of polymorphism because it permits a parent reference variable to assume many different forms. It can be a reference to a parent object, or it can be a reference to an instance of any class that inherits directly or indirectly from it.

Line 4 of the following code fragment illustrates the use of this form of polymorphism. Line 1 declares the variable `aBoat` to be a `Boat` class reference variable. Because the class `PowerBoat` (Figure 8.22) extends the class `Boat` (Figure 8.19), the variable can be morphed into a reference to a `PowerBoat` object. This is accomplished using the assignment statement on line 4. After line 4 executes, the variable `aBoat` stores the address of a `PowerBoat` object, the same object referenced by `pb1`.

```
//A super class reference variable can reference a subclass object
Boat aBoat; //declares a reference variable in the super class Boat
PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);

aBoat = pb1; //aBoat and pb1 now reference the same powerboat
```

Consistent with this form of polymorphism, a super class reference variable that refers to one type of subclass object can be reassigned to reference an instance of another one of its subclasses. After the following code sequence executes, the variables `aBoat` and `sb1` both reference the same `SailBoatV4` object, whose class (Figure 8.21) also extends `Boat`.

```
//A super class reference variable can reference any subclass object
Boat aBoat; //declares a references variable in the super class Boat
PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);
SailBoatV4 sb1 = new SailBoatV4(220, 200, 200, Color.GREEN, 200);

aBoat = pb1; //aBoat references a powerboat
aBoat = sb1; //aBoat now references a sailboat
```

NOTE *A super class reference variable can refer to any instance of a class that directly, or indirectly, inherits from it.*

Using the syntax of coercion, the address of a child class object stored in a parent reference variable can be coerced into a child reference variable. After line 5 of the following code fragment executes, the child reference variable `pb2` stores the address of a powerboat. If `pb2` were declared to be a reference to a subclass of `Boat` other than the `PowerBoat` class, line 6 would result in a translation error.

```
//A subclass reference can be coerced into a child reference variable
1   Boat aBoat; //declares a references variable in the super class Boat
2   PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);
3   PowerBoat pb2;
4
5   aBoat = pb1; //aBoat now reference the same PowerBoat child
6   pb2 = (PowerBoat) aBoat; //valid when aBoat references a PowerBoat
```

There is one restriction on the use of assignment statements that mixes child and parent reference variables. A child class reference variable cannot be assigned the address of a parent class object because a parent object is not a child object. An attempt to do so results in a translation error. A good way to remember this restriction comes from the old family adage: Parents can point to their children when they correct them, but it is rude for children to point to their parents.

The following code fragment uses the nonabstract parent class `RowBoat` defined in Figure 8.3 and the `SailBoat` class that extends it (Figure 8.6). Assigning the location of the parent class object declared on line 1 to the child class reference variable, `sb1`, on line 4 produces a translation error, as does line 5, which attempts to coerce the address of the parent `RowBoat` object into the child class reference variable.

```
//A subclass reference variable can NOT reference a superclass object
RowBoat rb1 = new RowBoat(30, 150, 200);
SailBoat sb1;

sb1 = rb1; //syntax error: child reference assigned a parent object
sb1 = (SailBoat) rb1; //coercion does not remedy the problem
```

NOTE *The addresses of parent objects cannot be assigned to child reference variables.*

8.5.2 Polymorphic Invocations

Definition

A **polymorphic invocation** is the act of invoking a method using a parent reference variable that refers to a child object.

When a method is invoked using a parent reference variable that refers to a child object, it is referred to as a **polymorphic invocation**. Consider the client code fragment shown in Figure 8.28 that uses the super class `Boat` (Figure 8.19) and its subclass `PowerBoat` (Figure 8.22). Although the variable `aBoat` declared on line 1 is of type `Boat,` it has been assigned the address of a child `PowerBoat` object. This is valid because parents can point to children. The `show` method is invoked on line 5 using this parent reference variable, which makes line 5 a polymorphic invocation of the `show` method.

```
1   static Boat aBoat = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);
2
3    public void draw(Graphics g)
4     {
5       aBoat.show(g);
6     }
```

Figure 8.28
Polymorphic invocation of the method **`show`** by the object referenced by **`aBoat`**.

The translator always looks into the class of the reference variable that invoked a nonstatic method to verify the signature of the method, for both polymorphic and nonpolymorphic invocations. As a result, line 5 begins its search in the class `Boat` for a method named `show` whose parameter list is a `Graphics` object. Because the `Boat` class contains a method with that signature (line 21 of Figure 8.19), line 5 is valid syntax. The parent class `Boat` need not implement the method `show`; it can simply define the method's name and its signature as an abstract method.

If the class `Boat` did not contain a `show` method whose parameter list matched line 5's argument list, the translator's search would progress *up* through the classes in `Boat`'s inheritance chain. Because the class `Boat` does not explicitly extend a class, the search would end unsuccessfully in the class `Object`, and line 5 of Figure 8.28 would produce a translation error.

NOTE *The class of the parent reference variable used in a polymorphic method invocation must include, or have inherited, an implementation or an abstract version of the invoked method.*

At runtime, the Java runtime environment uses a different starting point in its search to locate the method named in a polymorphic invocation. Unlike the translator that begins its search in the class of the parent reference variable coded in the invocation statement, the runtime environment begins its search in the class of the object referenced by the variable. For example, the search for the `show` method invoked on line 5 of Figure 8.28 would begin in the `PowerBoat` class because the variable `aBoat` references the `PowerBoat` object assigned to it on line 1. In effect, the invocation is morphed at runtime into an invocation of the `show` method that correctly draws a powerboat.

The programming language concept use to implement this form of polymorphism is called *dynamic binding*. The invocation is attached, or bound to, the method to be executed during runtime. Line 5 of Figure 8.29 is not a polymorphic invocation because line 1 declares `pb1` to be a `PowerBoat` reference variable and assigns it the location of a `PowerBoat` object. For non-polymorphic invocations, the method located by the translator's search to verify the existence of a method with the appropriate signature is the method executed at runtime.

```
1    static PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);
2
3    public void draw(Graphics g)
4    {
5      pb1.show(g);
6    }
```

Figure 8.29
Nonpolymorphic invocation of the method **`show`** by the object referenced by **`pb1`**.

The application `PolymorphicInvocations` shown in Figure 8.30 is a modification of the application `DesignTechniques` presented in Figure 8.23. This version of the application uses polymorphic invocations inside its `draw` call back method to produce the output shown in Figure 8.31, which is the same output produced by the original version (Figure 8.24).

Line 11 of Figure 8.30 creates three `Boat` reference variables, which are then assigned (lines 15–17) the location of the three objects created on lines 8, 9, and 10. This is valid syntax because all three of these objects' classes (Figures 8.20–8.22) extend the class `Boat` (Figure 8.19).

Because the variables `boat1`, `boat2`, and `boat3` now reference child objects, the invocations on lines 23–25 are polymorphic invocations. The appearance of a properly drawn rowboat, sailboat, and powerboat on the game board shown in Figure 8.31 confirms that the search path used by the Java runtime environment to locate the `show` method it executed began in the classes of the objects rather than the class of the three reference variables (i.e., `Boat`). If the search had begun in the `Boat` class, its `show` method would have drawn three hulls without oars, or a sail, or a propeller (lines 23-32 of Figure 8.19).

```
import edu.sjcny.gpv1.*;
import java.awt.*;

public class PolymorphicInvocations extends DrawableAdapter
{
  static PolymorphicInvocations ge = new PolymorphicInvocations();
  static GameBoard gb = new GameBoard(ge, "Design Techniques");
  static RowBoatV2 rb1 = new RowBoatV2(50, 200, 120, Color.YELLOW, 4);
  static SailBoatV4 sb1 = new SailBoatV4(220, 200, 200, Color.GREEN, 200);
  static PowerBoat pb1 = new PowerBoat(50, 300, 200, Color.MAGENTA, 400);
  static Boat boat1, boat2, boat3;

  public static void main(String[] args)
  {
    boat1 = rb1;
    boat2 = sb1;
    boat3 = pb1;
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    boat1.show(g);
    boat2.show(g);
    boat3.show(g);
  }
}
```

Figure 8.30
The application **PolymorphicInvocations**.

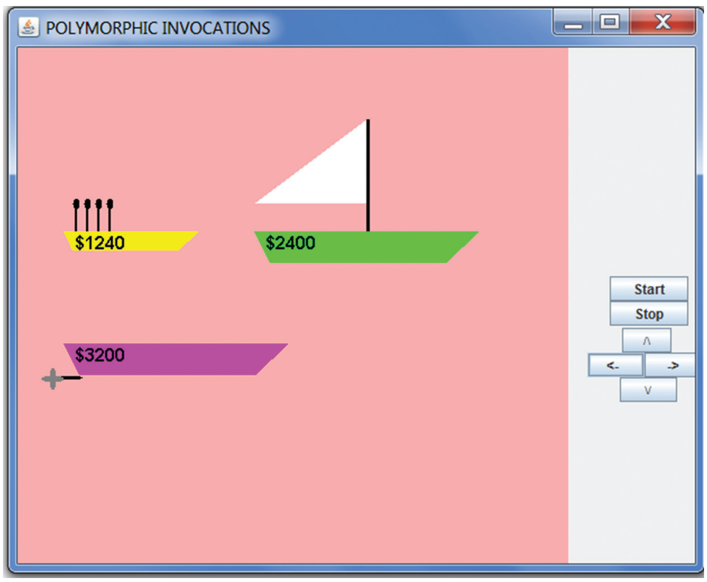


Figure 8.31
The output produced by the application **PolymorphicInvocations**.

8.5.3 Polymorphic Arrays

Definition

A **polymorphic array** is an array of parent reference variables that are used to store references to child objects.

Polymorphic arrays are declared using the syntax discussed in Section 6.5, which is used to declare any array of reference variables. For example, the array `inventory` declared below has the potential to become a polymorphic array because the class `Boat` defined in Figure 8.19 is the super class of the subclasses defined in Figures 8.20–8.22:

```
Boat[] inventory = new Boat[9];
```

If and when an element of the array `inventory` is assigned the address of a subclass object, it is then being used as a polymorphic array. Arrays of abstract class reference variables can only be used as polymorphic arrays because we cannot create an instance of an abstract class.

Each element of a polymorphic array could be assigned the address of the *same* type of subclass instance, in which case the array would be called a *homogeneous* polymorphic array. When at least two *different* subclass objects are referenced from within the array, the array is being used as a *nonhomogeneous* polymorphic array. Nonhomogeneous polymorphic arrays bring the power of arrays into the concept of inheritance and further reduce the number of lines of code required to produce an application.

The application `PolymorphicArrays` shown in Figure 8.32 is a modification of the application `PolymorphicInvocations` presented in Figure 8.30. It uses a polymorphic array to store a nine-boat inventory of Uncle Ed's boat store. The output it produces is shown in Figure 8.33.

Line 11 of Figure 8.32 creates a polymorphic array named `inventory` whose elements are reference variables in the abstract class `Boat` (Figure 8.19). Each time the loop that begins on line 15 executes, lines 17–21 create three new boats whose classes are shown in Figures 8.20–8.22. Then, lines 22–24 add them to Uncle Ed's inventory by storing them in the polymorphic array. The loop variable, `i`, is used on lines 17–20 to change the (x, y) location, number of oars, and size of the boats. It is also used on lines 22–24 to change the elements of the array that store the boats' addresses.

The `for` loop that begins on line 32 uses polymorphic invocations of the `show` method to draw each boat on the game board, which represents Uncle Ed's boat storage lot.

```
1   import edu.sjcny.gpv1.*;
2   import java.awt.*;
3
4   public class PolymorphicArrays extends DrawableAdapter
5   {
6     static PolymorphicArrays ge = new PolymorphicArrays();
7     static GameBoard gb = new GameBoard(ge, "POLYMORPHIC ARRAYS");
8     static RowBoatV2 rb;
9     static SailBoatV4 sb;
```

```java
    static PowerBoat pb;
    static Boat[] inventory = new Boat[9];

    public static void main(String[] args)
    {
      for(int i = 0; i < 3; i++)
      {
        rb = new RowBoatV2(10 + i * 130, 75, 120, Color.YELLOW, i * 2 + 2);
        sb = new SailBoatV4(10 + i * 170, 250, 110 + i * 15, Color.GREEN,
                            200 + i * 20);
        pb = new PowerBoat(20 + i * 160, 350, 120 + i * 15, Color.MAGENTA,
                           400);
        inventory[i * 3] = rb;
        inventory[i * 3 + 1] = sb;
        inventory[i * 3 + 2] = pb;
      }

      showGameBoard(gb);
    }

    public void draw(Graphics g)
    {
      for(int i = 0; i < 9; i++)
      {
        inventory[i].show(g);
      }
    }
  }
```

Figure 8.32
The application **PolymorphicArrays**.

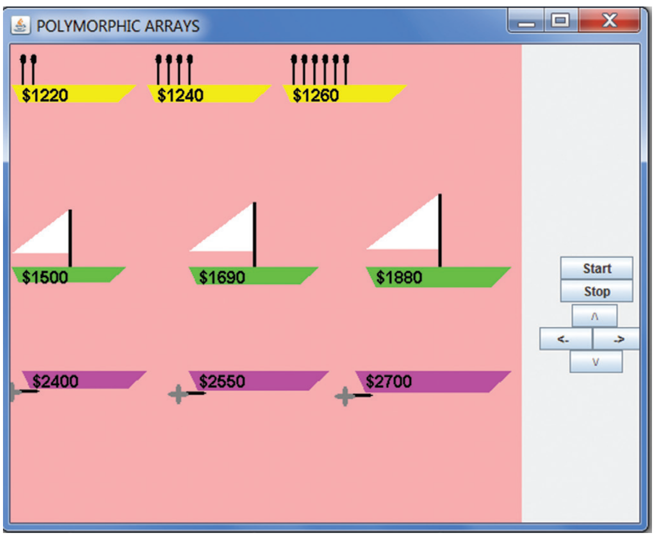


Figure 8.33
The output produced by the application **PolymorphicArrays**.

Advantages of Polymorphic Arrays

The addresses of the new boats created on lines 17–21 of Figure 8.32 could have been assigned directly into the array elements on these lines, in which case, lines 8–10 and 22–24 would be eliminated from the program. This approach would have reduced the program to 31 lines with much of the programs brevity coming from the use of a polymorphic array.

An alternative approach to this program would have been to declare three nonpolymorphic arrays, one for each type of boat, as shown below:

```
RowBoatV2[3] rb = new RowBoatV2[3];
SailBoatV4[3] sb = new SailBoatV4[3];
PowerBoat[3] pb = new PowerBoat[3];
```

The addition of these three lines and the additions to the body of the output loop (line 34), which now requires an output of three arrays rather than one, would expanded the 31-line version of the program to a 36-line program. Reductions in coding effort are typically realized when polymorphic arrays are used.

In addition, if the details of the nine boats in Uncle Ed's inventory, including their type, were to be input by the program user instead of being hard coded (as on lines 17–21), the polymorphic approach would reduce the storage requirements of the program. Any mix of nine rowboats, sailboats, and powerboats could be saved in the nine-element polymorphic array, but the three-nonpolymorphic-array approach would require that all three arrays be expanded to nine elements to accommodate the case when all of the boats are of one type (e.g., all sailboats).

8.5.4 Polymorphism's Role in Parameter Passing

Because a super class can point to (i.e., store the address of) an instance of any of its subclasses, a parameter whose type is the super class can be passed an argument whose type is any of its subclasses. Taking this concept to its extreme, because all classes inherit directly or indirectly from the class `Object`, any object's address can be passed to a parameter whose type is `Object`. We will utilize this fact when we code generic classes in Chapter 13.

Suppose we wanted to code a method that determined if two of Uncle Ed's boats occupied the same (x, y) location. We would code the method in the super class `Boat` (Figure 8.19) to enable any instance of its subclasses (a rowboat, a sailboat, or a powerboat) to invoke it. The method would compare that object's x and y data members to those of the objects passed to its parameter. To ensure that any of these three types of boats could be passed to the method, the parameter's type would be the super class `Boat`. The code for the method is given below:

```
public boolean samePosition(Boat aBoat) //code in the superclass Boat
 {
   if(x == aBoat.x && y == aBoat.y)
   {
     return true;
   }
   else
```

```
      {
        return false;
      }
    }
```

If the method was to determine if any of Uncle Ed's boats occupied the same (x, y) location, the static method shown below could be coded in the super class `Boat`. The method signature contains one parameter, a reference variable that can store the address of an array of instances of the class `Boat` or instances of `Boat`'s subclasses. When the method is invoked, it is passed the polymorphic array that contains Uncle Ed's boat inventory. It returns `true` if two or more boats are at the same (x, y) location.

```
public static boolean samePositionV2(Boat[] boats) //coded in the
{                                                  //superclass Boat
  for(int i = 0; i< boats.length; i++)
  {
    for(int j = i + 1; j< boats.length; j++)
    {
      if(boats[i].x == boats[j].x && boats[i].y == boats[j].y)
      {
       return true;
      }
    }
  }
  return false;
}
```

8.5.5 The methods `getClass` and `getName` and the `instanceof` operator.

Suppose your Uncle Ed expanded the requirements of his inventory program to include outputting the inventory of a specified type of boat. For example, print the details of all of the sailboats in the inventory. Then when a customer expressed an interest in a sailboat, Uncle Ed could give the customer a list of the sailboats currently in his inventory.

If the three nonpolymorphic arrays were used in the program, the type of the boat the customer was interested in could be used in the Boolean conditions of nested `if-else` statements to decide which of the three arrays to output. The following pseudocode fragment uses this approach to output the type of boat stored in the input string `typeSpecified`:

```
if(typeSpecfied.equalsIgnoreCase("rowboat")
{
    //a for loop to output the contents of the rowboat array
}
else if(typeSpecified.equalsIgnoreCase("sailboat")
{
    //a for loop to output the contents of the sailboat array
}
else
{
```

```
        //a for loop to output the contents of the powerboat array
    }
```

Although this implementation of the new requirement is rather straight forward, as discussed at the end of the previous section, this three-nonpolymorphic-array approach increases the program's size and its storage requirements. A more efficient single-polymorphic-array implementation of the new requirement could be used if there was some way of identifying the type each object referenced in the polymorphic array. The following pseudocode fragment could then be used to output only the type of boat stored in the input string `typeSpecified`:

```
for(int i=0; i < inventory.length; i++)//inventory is a polymorphic array
{
    if( //inventory[i] is of the typeSpedified )
    {
       System.out.println(inventory[i].toString());
    }
}
```

Fortunately, the API classes `Object` and `Class` provide the ability to identify the type, (i.e., the class) of any object.

The `getClass` Method in the Class Object

All classes inherit from the class `Object`. It is a super class of all classes contained in the API and the implied super class of every programmer defined class. The class `Object` is at the top of every class's inheritance chain. We have already taken advantage of this fact in Chapter 3 when we used the `toString` method inherited from the class `Object` to output the location of an object.

One of the other methods inherited from the class `Object` is its `getClass` method. This method has an empty parameter list and returns the location of an instance of a `Class` object.

The getName Method in the Class `Class`

When an object is constructed information about the object, such as its class name and the name of the class it extends, is recorded in an instance of a `Class` object that is created by the Java Virtual Machine and associated with the constructed object. The location of the associated `Class` object can be fetched by invoking `Object`'s `getClass` method on the constructed object, which can then be used to invoke any of the methods in the class `Class`.

One of these methods, `getName`, can be used to determine the name of any object's class. The method has an empty parameter list and returns a string containing the class's name. The following code fragment uses the five-parameter constructor of the class `PowerBoat` (Figure 8.22) to construct the object `pb2` and uses the `getClass` and `getName` methods to output the name of `pb2`'s class, `PowerBoat`:

```
PowerBoat pb = new PowerBoat(160, 350, 120, Color.MAGENTA, 400);
Class c = pb.getClass();
System.out.println(c.getName());
```

Often lines 2 and 3 are combined into one line:

```
System.out.println(pb.getClass.getName());
```

This abbreviated code version is used in the following code fragment to output only the `PowerBoat` objects stored in the polymorphic array `inventory`:

```
for(int i=0; i < inventory.length; i++)
{
 if(inventory[i].getClass().getName().equalsIgnoreCase("PowerBoat"))
  {
   System.out.println(inventory[i].toString());
  }
}
```

The application `ObjectAndClass` shown in Figure 8.34 is a modification of the application `PolymorphicArrays` presented in Figure 8.32. When launched, it asks the user what type of boat the customer is interested in and then uses the `getClass` and `getName` methods to identify that subset of Uncle Ed's inventory and outputs a description of these boats to the system console. The graphical and console output produced by the application is shown in Figure 8.35.

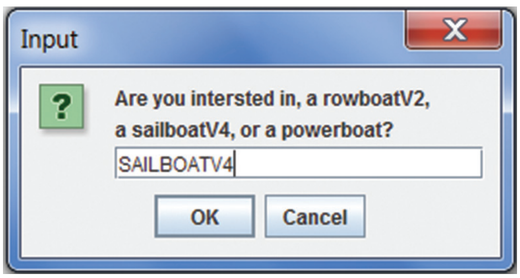
Line 8 of Figure 8.32 and the body of the loop that begins on line 15 create a polymorphic array named `inventory` whose elements are reference variables in the abstract class `Boat` defined in Figure 8.19. Each time the loop executes, lines 17–21 create three new boats whose classes are shown in Figures 8.20–8.22. The addresses of these objects are placed directly into the elements of the polymorphic array on lines 17, 19, and 21.

The Boolean condition of the *if* statement coded on line 29 uses the methods `getClass` and `getName` to determine if the class name of the object referenced by the `i`th element of the array `inventory` is the same as the string `s` input on line 25. If it is, the object is output to the console on line 31 using a polymorphic invocation of the `toString` method. The console output is shown in the bottom portion of Figure 8.35. Figure 8.35a shows the dialog box displayed by line 25 containing the user input *SAILBOATV4*, which resulted in the console output shown at the bottom of Figure 8.35. Figure 8.35b shows the graphical output produced by the polymorphic invocations of the `show` method (line 41 of Figure 8.34).

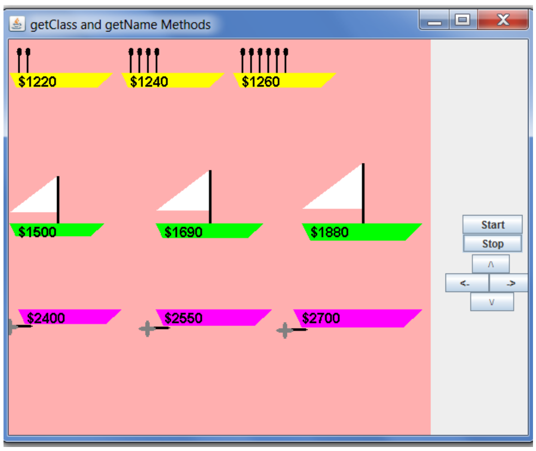
```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3    import javax.swing.*;
4
5    public class ObjectAndClass extends DrawableAdapter
6    { static ObjectAndClass ge = new ObjectAndClass();
7      static GameBoard gb = new GameBoard(ge, "getClass and getName Methods");
8      static Boat[] inventory = new Boat[9];
9
10     public static void main(String[] args)
11     {
12         String s;
13
```

```
      // Use of a polymorphic array
      for(int i = 0; i < 3; i++)
      {
        inventory[i*3] = new RowBoatV2(10 + i*130, 75, 120,
                                       Color.YELLOW, i*2 + 2);
        inventory[i*3 + 1] = new SailBoatV4(10 + i*170, 250 , 110+ i *15,
                                           Color.GREEN, 200 + i*20);
        inventory[i*3 + 2] = new PowerBoat(20 + i*160, 350 , 120+ i *15,
                                          Color.MAGENTA, 400);
      }

      s = JOptionPane.showInputDialog("Interested in, a rowboatV2," +
                                     "\na sailboatV4, or a powerboat?");
      for(int i = 0; i < inventory.length; i++)
      {
          if(inventory[i].getClass().getName().equalsIgnoreCase(s))
          {
            System.out.println(inventory[i].toString());
          }
      }
      showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 0; i < 9; i++)
    {
      inventory[i].show(g);
    }
  }
}
```

Figure 8.34
The application **ObjectAndClass**.



(a)



(b)

System Console Output:

Location: (10, 250), length: 110, Color: java.awt.Color[r=0,g=255,b=0], Sail Area: 200
Location: (180, 250), length: 125, Color: java.awt.Color[r=0,g=255,b=0], Sail Area: 220
Location: (350, 250), length: 140, Color: java.awt.Color[r=0,g=255,b=0], Sail Area: 240

Figure 8.35
The output produced by the application **`ObjectAndClass`**.

The instanceof Operator

Java provides a more succinct syntax than that used on line 29 of Figure 8.34 for determining the class of an object: its relational operator `instanceof`. This is a binary operator that can be used in a Boolean expression. Its first operand must be a reference variable, and its second operator must be a case-sensitive class name. The operator returns the value `true` when the class name is the class of the object referenced by the first operand. The Boolean condition on the last line of this code fragment evaluates to `true`.

```
Boat[] inventory = new Boat[2];
inventory[0] = new PowerBoat(50, 100, 200, Color.MAGENTA, 400);

if(inventory[i] instanceof PowerBoat)
```

Because a string variable cannot be used as one of the arguments, it cannot be used in the Boolean condition on line 29 of Figure 8.34 to determine if an element of the array `inventory` is an instance of the class whose name is stored in the string `s`.

The two most common uses of the `instanceof` operator are to:

1. Ensure that the casting of a polymorphic reference to an object does not result in a syntax error
2. Eliminate the need to include an implementation or an abstract version of a method in a parent class when the method is invoked polymorphically

The code fragment below demonstrates both of these uses. The `instanceof` operator is used on line 8 to prevent the translation error associated with an attempt to cast the `RowBoat` object declared in line 3 into a `PowerBoat` reference on line 10. In addition, because the casting permits a nonpolymorphic invocation of the `show` method on line 11, the translator looks into the `PowerBoat` class, rather than the `Boat` class, to verify the method's signature. The `Boat` class would not have to include a `show` method with the same signature as the `PowerBoat` class.

```
1   Boat[] inventory = new Boat[2]; //used as a polymorphic array
2   inventory[0] = new PowerBoat(50, 100, 200, Color.MAGENTA, 400);
3   inventory[1] = new RowBoatV2(50, 300, 75, Color.YELLOW, 2);
4   PowerBoat pb1;
5
6   for(int i = 0; i < inventory.length; i++)
7   {
8      if(inventory[i] instanceof PowerBoat) //show the powerboat(s)
```

```
        {
          pb1 = (PowerBoat) inventory[i];
          pb1.show()
        }
    }
```

NOTE *When using a preexisting super class to implement a polymorphic array, the use of the* `instanceof` *operator and casting eliminates the syntax error associated with using the elements of the array to polymorphically invoke a child class method whose signature is not defined in the parent class. It also eliminates the need to include a version of the method in the super classes that we write.*

8.6 INTERFACES

An **interface** is very similar to an abstract super class. Prior to Java Versions 8 and 9, they were similar to an abstract super class that contained only abstract methods and/or static final constant definitions. A basic understanding of interfaces is most easily explained by comparing them to this type of super class, so we will begin with a discussion of pre-Version 8 interfaces. The extended similarities introduced in Java Versions 8 and 9 will be discussed in Section 8.6.2.

Like a class, the source code of an interface is saved in a file with a `.java` extension, and its translation is stored in a file with a `.class` extension. In addition, it is good programming practice to begin the name of an interface with a capital letter.

Definition

An **interface** is a specification of the signatures of related methods that are implicitly abstract and/or a declaration of public constants that are implicitly static and final.

Figure 8.36 compares the syntax and keywords associated with an abstract super class and an interface. At the bottom of the figure, it also compares the syntax of heading of a class that *extends* an abstract super class and a class that *implements* an interface. As shown on the top right side of the figure, the heading of an interface substitutes the keyword **`interface`** for the two keywords **`abstract`** and **`class`** used in the definition of an abstract super class. In addition, the interface definition eliminates the keyword **`abstract`** used in the method signatures defined in an abstract class. Because the constant definitions included in an interface are implicitly **`static`** and **`final`**, these keywords are not used and the constants should be initialized. Interface methods and constant definitions are always implicitly **`public`**. They cannot be declared to be **`private`** or **`protected`**.

As shown in the bottom right side of Figure 8.36, a class that implements an interface uses the keyword **`implements`** in its heading rather than the keyword **`extends`**. Consistent with the use of these keywords, we say that a class implements an interface rather than extends it.

A syntactical difference not shown in Figure 8.36 is that while a class's heading can state that it extends one (and only one) super class, it can state that it implements more that one interface. When this is the case, the names of the interfaces in the class's **`implements`** clause are separated by commas. For example:

```
public class Class1 implements Interface1, Interface2, Interface3
```

<table>
<tr><th>Abstract Super Class
Definition of the super class <code>Parent</code></th><th>Interface
Definition of the interface <code>AnInterface</code></th></tr>
<tr><td>

```java
public abstract class Parent
{ public static final int a = 10;
  // … other constants

  public abstract int extra();
  //… other abstract methods

  public void show()
  { Sysem.out.println(this) }
  //… other non-static methods

  public static int add1(int a)
  { return a + 1; }
  //… other static methods
}
```

</td><td>

```java
public interface AnInterface
{ public int a = 10;
  // … other constants

  public int extra();
  // … other abstract signatures

  public default void show()
  { Sysem.out.println(this) }
  //… other non-static methods

  public static int add1(int a)
  { return a + 1; }
  //… other static methods
}
```

</td></tr>
<tr><td>Heading of a class that extends the super class Parent
<code>public class Child extends Parent</code></td><td>Heading of a class that implements the interface AnInterface
<code>public class AClass implements AnInterface</code></td></tr>
</table>

Figure 8.36
A comparison of abstract super class and interface syntax.

When a class implements one or more interfaces and also extends a class, the `extends` clause is coded in its heading before the `implements` clause. For example:

```java
public class Class2 extends Parent implements Interface1, Interface2
```

A class that includes an `implements` clause in its heading must implement all of the abstract methods whose signatures are included in the interface. There only two exceptions to this. The first exception is when the class is an abstract class, in which case it cannot implement any of the methods, and its subclasses must include implementations of all of the interface abstract methods. The second exception is when the class inherits implementations of the methods. These inherited implementations are treated like any other inherited methods. For example, the same search techniques are used to locate them at translation and runtime, and subclasses can override the inherited implementations. In addition, the inherited version of the overridden method can be invoked by preceding the method name with the keyword `super` followed by a dot.

As is the case with abstract classes, we cannot declare an instance of an interface, but the type of a reference variable can be the name of an interface. A very important use of interface reference variables and interfaces in general is in the coding of generic classes, which will be discussed in Chapter 13.

In summary, a pre-Version 8 interface can be considered to be an abstract super class that contains only public abstract methods and public static final constant definitions, with the following idiosyncrasies:

- a class can implement several interfaces
- interfaces and abstract classes have syntactical differences (shown in Figure 8.36)
- interfaces cannot contain method implementations
- interfaces are used in the coding of generic classes

When to Define and Use an Interface

The similarities of an interface and an abstract class can be a source of confusion when trying to decide which construct to use for a particular programming application. An interface is preferred when we want to standardize the signatures and functionality of methods that implement a commonly performed task on objects that may not be related. By "may not be related," we mean that with the exception of the class `Object`, the classes that perform these common tasks may not share a common ancestor.

For example, the need to compare two objects was anticipated to be such a common task that an interface named `Comparable` is included in the Java API. It defines the signature of a method named `compareTo`, and the interface's documentation describes the functionality of the method. Many nonrelated classes included in the API, such as the `String` class and the `BigInteger` class, implement this interface, and they all implement the functionality described in the interface's documentation. They all compare two objects and return a zero, a negative or a positive value, that reflects the equality or ordering of the two objects being compared. This use of interfaces facilitates the use of any class's `compareTo` method if we know that the class has implemented the interface `Comparable`. In general:

- an *interface* is defined to standardize the signatures and functionality of methods that implement a commonly performed task on objects that may not share a common ancestor other than the class `Object`
- interfaces facilitate the use of methods by standardizing both their signature and their functionality, as described in their documentation

The code of the interface `Drawable` is shown in Figure 8.37. It defines the signatures of two methods commonly used in game applications. As is typically the case, the description of the interface, which is given in Table 8.1, contains the signatures of the methods and describes the functionality of the two methods. It is considered good programming practice that all implementations of an interface's methods conform to the functionality described in the interface's documentation.

```java
import java.awt.*;

public interface Drawable
{
  boolean canDraw(int drawableWidth, int drawableHeight);
  void show(Graphics g);
}
```

Figure 8.37
The interface **`Drawable`**.

Table 8.1
The Documentation of the Interface Drawable

The Interface `Drawable`	
Methods	
Returned Type	**Signature and Functionality Description**
`boolean`	`canDraw(int width, int height)` Returns `true` if the object that invokes it can be drawn on a `Graphics` area whose lower right corner is at location (`width, height`)
`void`	`show(Graphics g)` Draws the game piece that invoked it on the graphic object `g` at its current (x, y) location

To close out our discussion of pre-Version 8 interfaces, consider the following scenario. The author of the game application `InterfaceUse`, shown in Figure 8.38, purchased a package containing the translated versions (bytecodes) of several game piece class implementations. The documentation of the package, which included a copy of Table 8.1, stated that all of the game-piece classes implemented the interface `Drawable` described in that table.

Six instances of two of these classes, `TopHat` and `SnowmanV9`, are declared on lines 11–16 of the application. Knowing that both of these classes implement the interface `Drawable`, the application programmer realized that the addresses of the six objects could be efficiently stored in a polymorphic array of type `Drawable` (which is declared on line 7 and assigned on lines 11–16 of Figure 8.38), and that the objects could then be efficiently drawn/not drawn using invocation of the two methods defined in the interface (lines 22–27 of Figure 8.38).

The output of the program is given in Figure 8.39, which reflects the fact that top hats can only be drawn if they are completely on the game board, and snowmen cannot be drawn if they are completely off the game board. The source code of the two game-piece classes and the class `GamePiece` that they extend (which in this hypothetical scenario, the application programmer never saw) are given in Figures 8.40, 8.41, and 8.42, respectively.

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3
4    public class InterfaceUse extends DrawableAdapter
5    { static InterfaceUse ge = new InterfaceUse();
6      static GameBoard gb = new GameBoard(ge, "INTERFACES", 700, 700);
7      static Drawable[] items = new Drawable[6];
8
9      public static void main(String[] args)
10     {
```

```
    items[0] = new TopHat(-10, 30, Color.BLUE, 51, 60); //part off
    items[1] = new TopHat(350, 360, Color.BLACK, 51, 60);
    items[2] = new TopHat(600, 640, Color.GREEN, 51, 60); //part off
    items[3] = new SnowmanV9(-10, 120, Color.BLUE, 80, 152); //part off
    items[4] = new SnowmanV9(200, 360, Color.BLACK, 80, 152);
    items[5] = new SnowmanV9(400, 640, Color.GREEN, 80, 152); //part off
    showGameBoard(gb);
  }

  public void draw(Graphics g)
  {
    for(int i = 0; i < items.length; i++)
    {
      if(items[i].canDraw(700, 700))
      {
        items[i].show(g);
      }
    }
  }
}
```

Figure 8.38
The application **InterfaceUse**.

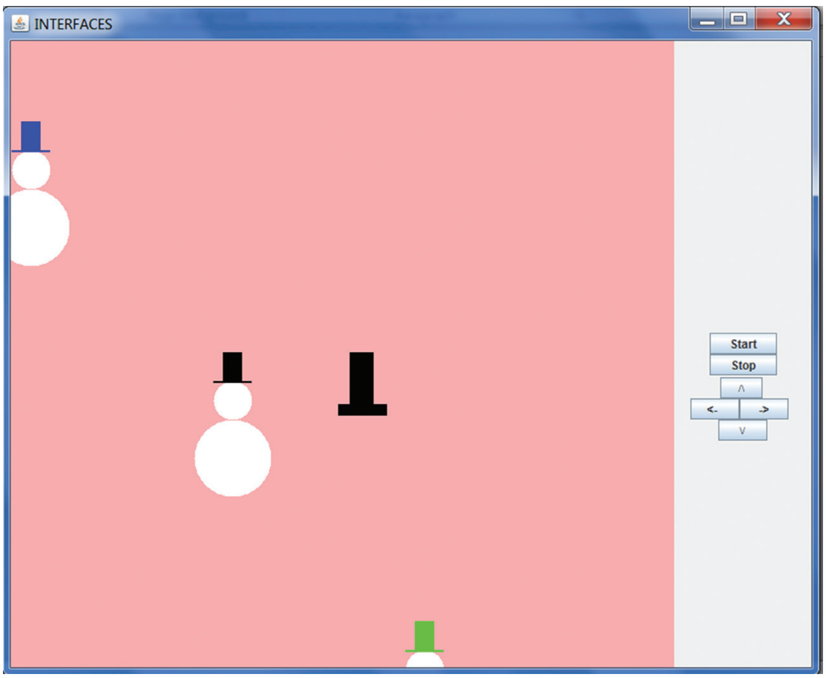


Figure 8.39
The output produced by the application **InterfaceUse**.

```
import java.awt.*;

public class TopHat extends GamePiece implements Drawable
{

  public TopHat(int x, int y, Color hatColor, int w, int h)
  {
    this.x = x;
    this.y = y;
    this.hatColor = hatColor;
    this.w = w;
    this.h = h;
  }
  public void show(Graphics g)
  {
    g.setColor(hatColor);
    g.fillRect(x + w/4, y, w/2, (int)(h*0.9));  // hat top
    g.fillRect(x, y + (int)(h*0.9), w, (int)(h*0.2));  // brim
  }
  public boolean canDraw(int gbWidth, int gbHeight) //Completely on the
  {                                                 //game board
    if(x >= 6 && x + w <= gbWidth
       &&
       y >= 30 && y + (int)(h * 1.1) <= gbHeight)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}
```

Figure 8.40
The class **TopHat**.

```
import java.awt.*;
import javax.swing.*;

public class SnowmanV9 extends GamePiece implements Drawable
{
  public SnowmanV9(int x, int y, Color hatColor, int w, int h)
  {
    this.x = x;
    this.y = y;
    this.hatColor = hatColor;
```

```
    this.w = w;
    this.h = h;
  }
  public void show(Graphics g)
  {
    g.setColor(Color.WHITE);
    g.fillOval(x + 20, y + 30, 40, 40); //head
    g.fillOval(x, y + 70, 80, 80); //body
    g.setColor(hatColor);
    g.fillRect(x + 30, y, 20, 30); //hat
    g.fillRect(x + 20, y + 30, 40, 2); //brim
  }
  public boolean canDraw(int gbWidth, int gbHeight) //Not completely off
  {                                                 //the game board
    if(x + w >= 6 && x <= gbWidth
       &&
       y + h > 30 && y <= gbHeight)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
}
```

Figure 8.41
The class **SnowmanV9**.

```
import java.awt.*;

public abstract class GamePiece
{
  protected int x;
  protected int y;
  protected int w;
  protected int h;
  protected Color hatColor;

}
```

Figure 8.42
The abstract class **GamePiece**.

As shown in Figures 8.40 and 8.41, the `TopHat` and `SnowmanV9` classes implement the interface `Drawable` and include an implementation of the functionality of the two `Drawable` methods particular to them. The implementer of the `TopHat` class decided that a top hat can only be drawn if it is completely on the game board, so the version of the method coded in Figure 8.40 returns true when this is the case. Snowmen will eventually be made to enter the game board from its edges, so they can be drawn if any part of them is on the game board. The code of the SnowmanV9's canDraw method coded in Figure 8.41 returns true when this is the case.

The output of the program shown in Figure 8.39 contains one top hat and three snowmen. This confirms the fact that the polymorphic invocations of the canDraw and show methods on lines 24 and 26 of Figure 8.38 are locating the correct subclass methods. As noted by the comments at the end of lines 11 and 13 of Figure 8.38, the blue and green top hats are partially off the game board, in which case their canDraw method returns false, and they are not drawn. All three snowmen are either partially or completely on the game board, in which case their canDraw method returns true, and they are all drawn.

The data members common to the classes TopHat and SnowmanV9 are collected in the abstract class GamePiece (Figure 8.42), which they extend. They are declared in this super class with protected access (lines 5–9 of Figure 8.42). As indicated in the top half of Figure 8.27, this gives the subclasses' methods the ability to access them directly without using set and get methods (e.g., lines 17 and 18 of Figure 8.40). In our hypothetical scenario, the application would be coded in a separate package, so it would not be able to access these data members.

8.6.1 Adapter Classes

When an interface contains a significant number of abstract methods, it is good programming practice to provide an adapter class for the interface. The term adapter class is a generic term for a class that implements an interface with methods that contain empty code blocks. It is also good programming practice to assign the name of the adapter class the name of the interface class it implements, concatenated with the word "Adapter." For example, the name of the adapter class for the interface Drawable shown in Figure 8.37 would be DrawableAdapter. The code of this class is given in Figure 8.43.

```
public class DrawableAdapter implements Drawable
{
    boolean canDraw(int drawableWidth, int drawableHeight)
    {
        return false;
    }
    void show(Graphics g);
    {

    }
}
```

Figure 8.43
The adapter class **DrawableAdapter**.

The adapter class is provided to permit a new class to implement only the methods defined in the interface that are relevant. When this is the case, the new class extends the adapter class and then overrides the empty methods with its implementation. For example, if the specification of the class `RocketShip` only required that it implement the `show` method in the `Drawable` interface, then, assuming the `DrawableAdapter` class shown in Figure 8.43 exists, the code of the `RocketShip` class would be written as follows:

```
public class RocketShip extends DrawableAdapter
{
  // the data members of the class RocketShip

  @Override
  public show (Graphics g)
  {
    //the code to draw a RocketShip object
  }

  //the remainder of the methods of the class RocketShip
}
```

We will extend some of the adapter classes in the Java API when we study graphical user interfaces in Chapter 11. The game programming environment contains an interface named `Drawable` that defines the signatures of all of the `draw` call back methods described in Appendix A. The game environment also contains the adapter class `DrawableAdapter`, so a game program's class does not have to implement all of the call back methods if it extends `DrawAbleAdapter`, as does line 4 of Figure 8.37.

It should be noted that when a new class is a child class the use of an adapter class is not an alternative, because a class can only extend one class. The new class's heading would have to contain an **`implements`** clause, and all of the interface methods would have to be implemented. The interface methods not used by the class would simply contain an empty code block.

8.6.2 Interface Enhancements in Java Versions 8 and 9

An interface's similarity to abstract classes was expanded in Java Versions 8 and 9 beyond the inclusion of abstract method signatures and final static constants. Like an abstract class an Interface can now also contain *public* static and non-static method implementations, which the implementing class effectively inherits.

As shown in the bottom half of Figure 8.36, the syntax of a non-static interface method must include the keyword **`default`** in its signature, and the syntax of a static interface method is the same as that used within an abstract class. As implied by the keyword **`default`**, the interface's implementation of a non-static method can be overridden within a class that implements the interface. Unlike abstract classes, interfaces can*not* contain declarations of data members (non-static class level variables).

The syntax used to access static final constants and invoke static and default methods defined in an interface, is the same syntax used to access static final constants and invoke static and default methods defined in an abstract class. The only exception is the syntax used to invoke a default method that has been overridden by the implementing class. Simply preceding the method name with the prefix `super.`, as we would to invoke an overridden method defined in a super class, is an ambiguous syntax because a class can implement more than one interface. To resolve this ambiguity, the name of the interface that contains the version of the method being invoked is used as a prefix to `super.` in the invocation.

For example, if signature of the overridden default method being invoked was `void aMethod( )` and it was coded in the interface `Interface3`, the invocation would be:

```
Interface3.super.aMethod();
```

The keyword `super` is included in the syntax to distinguish the invocation from that of a static method invocation.

An Interfaces Role in the Design Process

An interface's role in the design process is to provide implementations of methods commonly used within the classes of many different applications by simply including an implements clause in the application's class headings. Since a class can implement more than one interface, these methods can be coded within several different interfaces. The implementing class can utilize, and possibly override the method implementations contained in the interfaces, and still be able to declare itself to be a child of another class. In addition, the design of a particular application can include a new interface that defines static final constants, abstract methods, and static and default method implementations common to several of its classes that are not already defined in existing interfaces.

However, an interface named `Boat` could not replace the abstract super class `Boat` in the design presented in Figure 8.18, because an interface cannot contain data members. The four data members (`x`, `y`, `length`, and `color`) defined in the class `Boat` would still have to be defined in a parent class, or within each of the other three classes. And since an interface, like a parent class, cannot directly access data members defined in its implementing classes, the interface `Boat` could not contain the `get` and `set` method implementations contained in the class `Boat`. These too would still have to be defined in a parent class, or within each of the application's classes.

But a method in an interface could invoke the `get` and `set` methods coded in the classes that implement the interface using the same coding techniques that permit parent class methods to invoke child class methods previously discussed in Section 8.4.2. Thus the `Boat` class's `calculatePrice`, `show`, and `toString` methods in Figure 8.19 could be coded in an interface. Their code would invoke the `get` methods coded in its implementing classes to fetch the data members declared within its implementing classes, or declared within these classes's parent class. Lines 23–26 of the `show` method in Figure 8.19 illustrate this coding technique.

8.7 SERIALIZING OBJECTS

Definition

Object serialization is the act of disassembling objects before writing them to a disk file.

Object deserialization is the act of reassembling objects after they are read from a disk file.

In Section 4.8, we discussed techniques used to transfer information to and from a disk file. As part of these techniques, information is written to the disk using the methods in the `PrintWriter` class (e.g., `println` and `print`). These methods write a string to the file, which means that the information in the file is represented as ASCII characters. Any piece of information written to the file that is not a string must be converted to a string before it is written.

This means that when the contents of the variables `houseNumber` and `quantity` defined in the code fragment below are written to the disk, they produce the same output to the disk: the three characters *175*.

```
String houseNumber = "175"
int quantity = 175;
```

Because the type of the information written to the file (e.g., `String` and `int)`is not represented in the file, a written description of the information in the file must be provided with the file to properly read and process the data in the file. When a reference to an object is included in the parameter passed to the `PrintWriter`'s output methods, the object's `toString` method is implicitly invoked, and the returned string is written to the file. In this case, the file description should include not only the types of the data members written to the file, but also the order in which they were written and the objects' classes, so the object written to the file can be properly reconstructed by the program that reads the information from the file.

The `writeObject` method in the class `ObjectOutputStream` presents a better alternative for writing the data members of an object to a disk file. When this method is used to write an object to a file, all of the information required to reconstruct the object when it is read from the file (the data members' types, the order in which they were written, and the class of the objects) is written to the file by the method. This information can then be used by the `readObject` method contained in the `ObjectInputStream` class to reconstruct the object when the method is used to read the object from the file. The gleaning of all of this additional information from objects when they are written to a file is called **object serialization**, and the use of this information to efficiently recreate objects when they are read from a file is called **object deserialization**.

Object serialization allows us to write all of an object's data members to a disk file as a single entity by simply invoking the `ObjectOutputStream` class's `writeObject` method and passing the object to the method's parameter. Object deserialization allows us to read all of an object's data members from a disk file by simply invoking the `ObjectInputStream` class's `readObject` method. This method recreates a serialized object and returns the address of the recreated object. Because the class of each object is written to the file, the file can contain different types of objects. When these objects are related as subclasses of the same super class, the file can be written to and read from polymorphically.

The application `SerializingObjects` presented in Figure 8.44 demonstrates the serialized writing and reading of objects to/from a disk file. It is a modification of the application `PolymorphicArrays` shown in Figure 8.32. This version of the application outputs the nine-boat inventory of Uncle Ed's boat yard, stored in a polymorphic array, to a serialized disk file using the `writeObject` method and reads the objects back into the array using the `readOb-`

`ject` method. Sample outputs produced at various points in the program's execution are given in Figure 8.45.

Line 12 of Figure 8.44 creates a polymorphic array named `inventory` whose elements are reference variables in the abstract class `Boat` (Figure 8.19). Each time the loop that begins on line 16 executes, lines 18–22 create three new boats whose classes are shown in Figures 8.20–8.22. Then, lines 23–25 add them to Uncle Ed's inventory by storing them in the polymorphic array. When the program in Figure 8.44 is launched, line 37 of the `draw` method displays the boats on the game board because each element of the array `inventory` contains a non-null reference (line 35). The initial output of the program is shown in Figure 8.45a.

When the game board's right arrow button is clicked, line 59 of the button's call back method writes the serialized version of all of the boat objects to a disk file named "Inventory." The file is created and attached to the `ObjectOutputStream` object `outFile` on lines 52 and 53; this is the object used on line 59 to invoke the `writeObject` method.

After subsequent clicks of the game board's up arrow button, line 46 of the button's call back method deletes the boats from the polymorphic array by setting all of its elements to `null`. When the call back method ends, the game environment invokes the `draw` call back method. This method then displays an empty game board (Figure 8.45b) because line 35 prevents the `show` method from executing. If it had executed, the program would have been terminated by a `NullPointerException` because all of the elements of the array `inventory` are `null`.

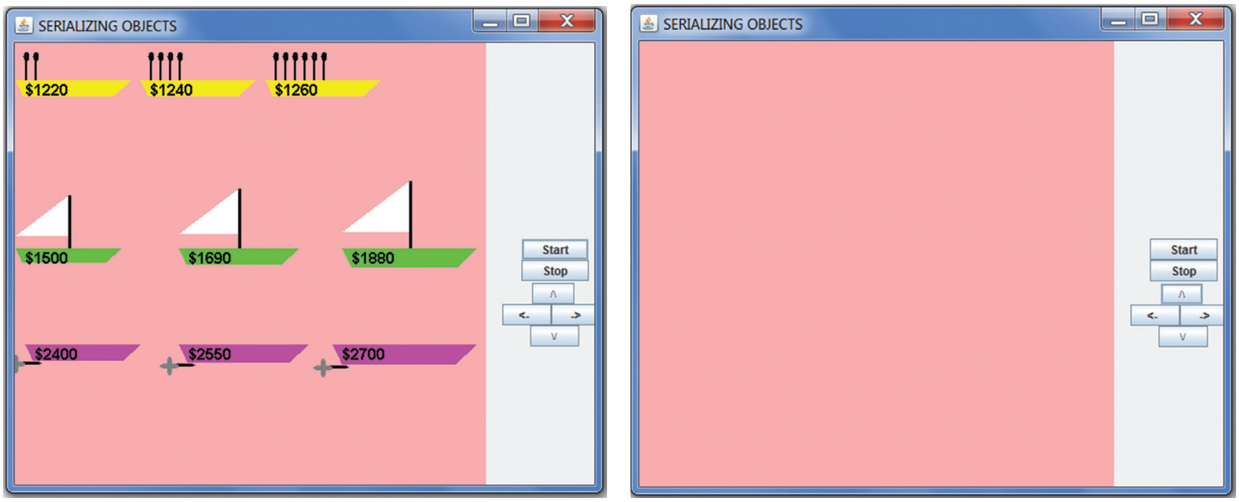
To restore and redisplay Uncle Ed's inventory, line 77 of the left button call back method reads the serialized boat objects from the disk file "Inventory" and places their addresses into the polymorphic array. The file is opened and attached to the `ObjectInputStream` object `inFile` on lines 72 and 73; this is the object used on line 77 to invoke the `readObject` method. After the left button call back methods ends, the `draw` call back method executes and line 37 of the `draw` call back method displays the restored nine-boat inventory on the game board (left side of Figure 8.42) via polymorphic invocations to the boats' `show` methods. The redisplay of the inventory verifies that line 59 correctly wrote the serialized objects to the disk file, and line 77 correctly read them back into the polymorphic array.

The coercion used in Figure 8.44 on line 77 is necessary because the `readObject` method, invoked on that line, returns a reference whose type is `Object`. When writing and reading serialized objects, the exceptions coded on lines 64, 81, and 84 must be caught (as shown) or thrown (to be discussed in Chapter 10) inside the methods that perform the disk I/O. Alternately, a `throws` clause can be included in the methods' headings. If the methods override a method, the overridden method must contain the same `throws` clause, or the `throws` clause alternative cannot be used. Because the call back methods override the methods in the `DrawableAdapter` class, the `catch` clause alternative was used in this program.

```
import edu.sjcny.gpv1.*;
import java.awt.*;
import java.io.*;

public class SerializingObjects extends DrawableAdapter
{
 static SerializingObjects ge = new SerializingObjects();
 static GameBoard gb = new GameBoard(ge, "SERIALIZING OBJECTS");
 static RowBoatV2 rb;
 static SailBoatV4 sb;
 static PowerBoat pb;
 static Boat[] inventory = new Boat[9];

 public static void main(String[] args)
 {
  for(int i = 0; i < 3; i++)
  {
   rb = new RowBoatV2(10 + i * 130, 75, 120, Color.YELLOW, i * 2 + 2);
   sb = new SailBoatV4(10 + i * 170, 250, 110 + i * 15, Color.GREEN,
             200 + i * 20);
   pb = new PowerBoat(20 + i * 160, 350, 120 + i * 15, Color.MAGENTA,
            400);
   inventory[i * 3] = rb;
   inventory[i * 3 + 1] = sb;
   inventory[i * 3 + 2] = pb;
  }

  showGameBoard(gb);
 }

 public void draw(Graphics g)
 {
  for(int i = 0; i < 9; i++)
  {
   if(inventory[i]  != null)
   {
    inventory[i].show(g);
   }
  }
 }

 public void upButton() //delete the RAM based inventory
 {
  for(int i = 0; i < 9; i++)
   {
    inventory[i] = null;
   }
 }
```

```
  public void rightButton() //output inventory to the file
  {
    try
    { FileOutputStream fos = new FileOutputStream("Inventory");
      ObjectOutputStream outFile = new ObjectOutputStream(fos);

      for(int i = 0; i < 9; i++)
      {
        if(inventory[i] != null)
        {
          outFile.writeObject(inventory[i]);
        }
      }
      outFile.close();
    }
    catch(IOException e)
    {
    }
  }

  public void leftButton() //input inventory from the file
  {
    try
    { FileInputStream fis = new FileInputStream("Inventory");
      ObjectInputStream inFile = new ObjectInputStream(fis);

      for(int i = 0; i < 9; i++)
      {
        inventory[i] = (Boat) inFile.readObject();
      }
      inFile.close();
    }
    catch(IOException e)
    {
    }
    catch(ClassNotFoundException e)
    {
    }
  }
}
```

Figure 8.44
The application **SerializingObjects**.



(a) (b)

Figure 8.45
Outputs produced by the application **`SerializingObjects`**.

In order for an object to be serialized, its data members must be serializable. Primitive variables and strings are serializable, as are primitive arrays. The API documentation indicates that if a class implements the interface `Serializable`, then instances of the class are serializable; in addition, objects whose data members reference these serializable objects are serializable. The documentation of the API classes state which interfaces they implement. The API `Color` class, the `BigDecimal` and `BigInteger` classes, and many other API classes implement the interface `Serializable`.

The heading of the class whose objects are permitted to be serialized should indicate that it implements the interface `Serializable`, and all of its data members should be serializable. If any of the class's data members are not serializable, they should be declared `transient`.

The `Serializable` interface is imported from the API `java.io` package (line 3 of Figure 8.44), and it does not contain any method signatures to implement. Because all of the objects being serialized in the application `SerializingObjects` extend the class `Boat`, the **`implements`** clause was added to that class's heading (line 4 of Figure 8.19).

8.8 JAVA DOCS

Javadoc is a tool used to standardize and automate the documentation of Java constructs such as classes, interfaces and packages and the entities within them such as data members, constructors, methods, and inner classes. The documentation produced is in an HTML format. Javadoc is the tool used to produce the on-line documentaiontion of the Java API. We can use this tool to rapidly produce documentation in a format familure to Java programmers for the classes, interfaces, and packages we write.

The Javadoc tool can be downloaded, and is incorporated into Java integrated development environments such as Eclipse and NetBeans. Figure 8.46 shows the first two portions of a Java Doc that was created within the NetBeans IDE. It documents the class `Student` shown in Figure 8.47, which was coded within that environment. The left and right side of the figure show the first and second portion of the generated documentation respectively. For brevity, the remainder of the documentation is not shown.

Class Student

java.lang.Object
  Academic
    Student

All Implemented Interfaces:
`Gradeable`

```
public class Student
extends Academic
implements Gradeable
```

An instance of this class exists within an Academic environment, and is Gradeable

Field Summary

Fields

Modifier and Type	Field and Description
`java.lang.String`	`address` A student's address
`java.lang.String`	`name` A student's name: last, first
`protected static double`	`tuition` A student's annual tuition

Fields inherited from interface Gradeable

`weeksPerSemester`

the Class and Field portion of the Java Doc

(a)

Constructor Summary

Constructors

Constructor and Description
`Student()` Creates a student with a default name Jones
`Student(java.lang.String name, java.lang.String address)` Creates a student with the specified name and address

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
`java.lang.String`	`getName()` Gets a student's name
`protected void`	`setAddress(java.lang.String address)` Sets a Student's address

Methods inherited from class Academic

`parentMethod`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString,`

the Method portion of the Java Doc

(b)

Figure 8.46
The first two portions of the Java Doc of the class `Student` shown in Figure 8.47.

After the entity to be documented is written within a programming environment a few simple clicks initiates the generation of the default Javadoc HTML file, which is saved in the project's folder and then displayed in the default browser. For example in NetBeans the file would be generated by clicking *Run* on the main menu, and then clicking *Generate Javadoc (Java Docs Generator)* within the drop-down menu. The HTML file would be placed in the project folder's subfolder `\dist\javadoc`. Private methods and data members are not included within the generated documentation.

```java
/**
* An instance of this class exists within an Academic
* environment, and is gradable
*/
public class Student extends Academic implements Gradeable
{
   /** A student's name: last, first */
   public String name;
   /** A student's address */
   public String address;
   /** A student's annual tuition */
   protected static final double tuition = 14300.00;

   /** Creates a student with a default name Jones */
   public Student()
   {
      name = "Jones";
   }

   /** Creates a student with the specified name and address
   * @param name The student's name: last, first
   * @param address The students address
   */
   public Student(String name, String address)
   {
      this.name = name;
      this.address = address;
   }

   /** Gets a Student's name
   * @return A string representing the student's name
   */
   public String getName()
   {
      return name;
   }

   /** Sets a Student's address
   * @param address The student's address
   */
   protected void setAddress(String address)
   {
      this.address = address;
   }
}
```

Figure 8.47
The class `Student` containing Javadoc comments included in Figure 8.47a and b.

To add additional text to the default documentation of an entity (e.g. a class, a method, etc.), a multi-line Javadoc comment containing the additional text is added immediately before the entity's code in the Java source file. These comments begin with the three sequential key strokes `/**` and end with the two sequential keystrokes */. The class `Student` shown in Figure 8.47 contains eight Javadoc comments. The comment on lines 1–4 contains two lines of additional documentation text, lines 2–3, each of which begins with an asterisk. These leading asterisks are optional and do not become part of the Java Doc, but most programmers feel they improve the readability of the Java source code.

The text within each entity's comment is incorporated into the portion of the Java Doc where the entity's documentation appears. For example, the text on lines 2–3 within the comment that precedes the class's heading is made part of the Class documentation near the top of Figure 8.46a. Similarly the text within the comments that precede each of the class's data member declarations, lines 7–12, is added to the Fields (data member) area of Figure 8.46a. At the bottom of that figure a static field `weeksPerSemester`, which Javadoc gleaned out of the interface `Gradeable` that `Student` implements, is shown.

Figure 8.46b shows the next part of the Java Doc that describes a class's constructors and methods, followed by the class's inherited methods. The text in the comments that begin on lines 14, 20, 30, and 38 of Figure 8.48 are displayed in the appropriate areas of this part of the Javadoc. The text in first line of these comments that do not begin with an at-sign (`@`), are displayed in the portion of the documentation shown in Figure 8.46b. Comment text that begins with an at-sign, when included, is coded immediately after the text that does not begin with an at-sign.

Javadoc includes 11 tags that begin with an at-sign. The two used in Figure 8.47, `@parm` (on lines 21, 22, and 39) and `@return` (on line 31), are the most often used tags. They add programmer composed descriptions of a method's parameters and its returned type to the last (Detail) portion of the Java Doc, which is not shown in Figure 8.46. The descriptive text is written after the tag. When the `@parm` tag is used, the name of the parameter being described must appear between the tag and the descriptive text. Below is the Method Detail area of the Detail portion of the Java Doc that includes the text after the tags on lines 31 and 39.

Method Detail

- **getName**
  `public java.lang.String getName()`
  Gets a student's name
  Returns:
  A string representing the student's name

- **setAddress**
  `protected void setAddress(java.lang.String address)`
  Sets a Student's address
  Parameters:
  `address` - The student's address

8.9 CHAPTER SUMMARY

Inheritance establishes a parent-child relationship between two classes in which all of the methods and data members of the parent class become part of the child class. Including the keyword **`extends`** in the class's heading of any Java class designates it as a child class, and the class name that follows it designates its parent class. A Java class can have one parent, and it can have multiple children unless the class is declared to be **`final`**. A child class also inherits the data members and methods of its parent's class's parent recursively. When a method is invoked on an object, the runtime environment begins its search for the method in the object's class and continues the search up the class's inheritance chain.

Functionality is added to an inherited class by coding additional data members and methods in the child class and by overriding an inherited method: that is, including a method in a child class that has the same signature as an inherited method. The translator will verify the signature of an overridden method if the child class includes the `@Override` directive before the signature of its version of the method. This is considered to be good programming practice. Inherited methods can be overloaded in the child class by changing the type or number of parameters in their parameter list.

To prevent a method from being overridden, the keyword **`final`** is used in its signature. When a class is declared to be **`final`**, it cannot be extended. Methods that enforce security on systems are usually declared to be **`final`** to prevent them from being overridden. An abstract class is used to collect data members and methods that are common to several classes into one parent class, which is considered to be good design practice. A class is designated to be abstract by including the keyword **`abstract`** in its heading before the keyword **`class`**. An abstract class cannot be instantiated.

Parent class constructors are not inherited, but they can be invoked as the first line of a child class constructor by coding the keyword **`super`** followed by an argument list. If the child does not explicitly invoke a parent constructor, the parent's default constructor is implicitly invoked. The child can also invoke inherited **`public`** or **`protected`** methods, including parent methods that were overridden, by beginning the invocation with the keyword **`super`** followed by a dot. This is often done in overridden methods to include the functionality of the parent method in the child's version of the method. Methods and data members whose access is designated **`protected`** are hidden from non-child classes contained in a separate package. Protected methods and data members of a class can be accessed by members in the same package as the class as well as by members in its subclass.

Polymorphism is a powerful feature of inheritance that is based on the fact that parent reference variables can store the address of a child class object. This permits us to store references to any type of child object in one array of parent reference variables, and to execute each child's version of a method by invoking it on each element of the polymorphic array. It also permits a child class instance to be passed to, and processed by, a method whose parameter type is that of the parent class. The method can perform child class specific processing by using the `getClass` and `getName` methods and the `instanceof` operator to determine the name and class of the object passed to the method. Alternately, polymorphism also permits a parent method to invoke a child class method to perform child class specific processing. The parent class usually includes an abstract version of the method to impose a translator enforced requirement that its children provide an implementation of the method.

`Interfaces` are similar to `abstract classes` that only contain abstract methods and static final constants. Interfaces are used to share constants and to standardize the signatures and functionality of methods that perform common tasks on objects that may not share a common ancestor. When a class includes an **`implements`** clause in its heading, it must provide an implementation for every method defined in the interface. Alternately, it could extend a class (referred to as an `adapter class`) that provides empty implementations of the interface's methods and override one or more of the methods. A class can implement multiple interfaces, and interfaces play an important role in the implementation of generic methods and classes.

Objects can be transferred to and from disk files using the techniques called serializing and deserializing, respectively. Object serialization is the act of disassembling objects before writing them to a disk file, and object deserialization reassembles them after they are read from the disk file.

Knowledge Exercises

1. True or false:

a) Parent class is to child class as base class is to super class.

b) A parent class can inherit data members and methods from to a child class.

c) A Java parent class can have multiple child classes that inherit its data members and methods.

d) A parent class includes an **`extends`** clause in its heading to specify its child class.

e) A child class inherits all the methods of its parent class including the parent's constructors.

f) In Java, a child class may only extend (inherit from) one class.

g) Several child classes can inherit from the same parent.

h) Parent class constructors are not inherited, but they can be invoked from within the child class.

i) When two methods in an inheritance chain have the same name but different parameter lists, we say they are overridden.

j) An inherited method can be invoked by a method within a child class.

k) The keyword **`final`** is used to prevent a method from accidentally being overridden.

l) A class can implement more than one interface.

m) To extend a class, we must have the source code of the class.

n) By default, a parent's no-parameter construct is always invoked when a child class constructor begins execution.

2. Explain the difference between chain inheritance and multiple inheritance. Which one does Java support?

3. What are two advantages of using inheritance in the software design and implementation processes?

4. Give the statement used in a child class method to invoke an inherited two-parameter constructor.
5. Explain the restrictions imposed by these three types of access: public, private, and protected.
6. Give the statement used in a child class method to invoke an inherited method named `output` that has no parameters and is not overridden.
7. Give the statement used in a child class method to invoke an inherited method named `input` that has no parameters and is overridden.
8. Why should the `@Override` directive be used, and where is it coded?
9. How can we prevent a method from being overridden by a child class?
10. Give two reasons for overriding an inherited method.
11. Explain when you would overload a method instead of overriding it.
12. How can we prevent a class from being extended? Give an example of when we would want to impose this restriction.
13. Explain how an abstract class is used during the design process.
14. True or false:
    a) A parent reference variable can store the address of a child object.
    b) A child reference variable can store the address of a parent object.
    c) The address of a child object stored in a parent reference variable can be assigned to a child reference variable.
    d) When a method is invoked on a parent reference variable, the version of the method in the parent's class always executes.
    e) Child references variables can be passed to parent type parameters.
15. Give the declaration of a 200-element polymorphic array that can store instances of the child classes `Train` and `Airplane` that extend the class `Transporter`.
16. What are the differences between using **`extends`** and **`implements`** in a class's heading?
17. What can be included in an interface, and what is an advantage interfaces have over abstract classes?
18. What is an adapter class, and what is the advantage of extending an adapter class?
19. Explain what you would do to implement an adapter class for the interface `ManyMethods` that defines 20 method signatures.

Programming Exercises

1. Develop a UML diagram for a class named `Vehicle` that has three private data members: `price`, `color`, and `model`. Its methods will include a default and a three-parameter constructor, a `toString` method, `set` and `get` methods, and a method to input all of the values of its data members. Assume the class `Vehicle` extends the class `Transporter`.

2. Implement the class described in Exercise 1 and write an application that verifies your implementation. You can assume the `Transporter` class has no data members or methods.

3. Cars and trucks have a price, color, and model. Cars also have a radio type, and trucks have a maximum tonnage that they can haul. Develop the UML diagrams for a car and a truck class using the concepts of inheritance to reduce the time and effort required to develop the classes. All of the data members should be private, and each class should have a four-parameter constructor, a `toSting` method, `set` and `get` methods, and a method to input all of the values of its data members.

4. Implement the classes described in Exercise 3 and write an application that verifies your implementation. Then, change the application so it can be used to input a mix of 10 cars and trucks and output them to the system console.

5. Modify Exercise 4 to include the use of a polymorphic array.

6. Write an application that asks the user how many two-dimensional shapes (squares, rectangles, circles and ellipses) to draw on the game board, as well as the type, location, and color of each shape. Any of the circles or squares can be drawn filled or unfilled. After all inputs have been performed, the shapes will appear on the game board. The name of each shape will be displayed just above it, as will the formula to compute the shapes' area if the shape is a circle or a rectangle. When user strikes the S, R, C, or E key, all of the squares, rectangles, circles, or ellipses will alternately appear or disappear respectively. Your design should utilize the techniques of inheritance and polymorphism to minimize the effort required to produce the program.

7. Supermarkets carry three categories of items in their inventory: canned goods, flowers, and produce. Every item in the store has a name, a unit price, and a quantity in stock. In addition, produce items have an expiration date and a weight (because they are sold by the pound), and flower items have a color and a variety (i.e., house plant, garden plant, arrangement).

   Develop UML diagrams for each of the three categories of items, canned goods, produce, and flowers, utilizing the techniques of inheritance and polymorphism to minimize the effort required to implement the classes, then implement the classes and verify their implementation. All of the data members should be private, and each class should have a constructor that can be passed the values of all of its data members, a `toString` method, `set` and `get` methods, a method to input all of the values of its data members, and a method to compute an item's price.

   The price of an item is its base price plus a 15% profit margin. The base price of a canned-good item is just its unit price. The base price of a produce item is its unit price times its weight, and the base price a flower item is its unit price, except for arrangements which have a $5.00 preparation fee.

8. Design and implement a class called `SuperStore` that clients can use to declare a store that sells the canned goods, produce, and flowers described in Exercise 7. When an instance of this class is created, its constructor will be passed the number of items (canned goods plus produce plus flower items) that the store will carry, and the city location of the store (a string). The class will contain:

a) a method (named `Superstore(maxNumOfItems, city)`) to create an instance of a super store and specify its location and the maximum number of items the store will carry;

b) a method (named `addItem(anItem)`) to add a new canned good, produce, or flower item to the store's inventory (one method);

c) a method (named `outputInventory()`) to output all the information for the entire inventory (all canned goods, produce, and flowers) to the system console preceded by the store's name and location,

d) a method (named `outputGenericGroup(integerCategoryID)`) to output all the information for the entire inventory of either canned goods, produce, or flowers to the console, preceded by the store's location. Your design should include a polymorphic array to store the Superstore's inventory.

Note: Whenever an item is output, the output should include all of an item's input information, preceded by the item's generic category (e.g., *This item is a Canned Good*) and the item's calculated selling price.

9. Write an application that creates an instance of the superstore described in Exercise 8, allows the user to enter the initial inventory (items a–d below), and then repeatedly outputs the inventory (item e below).

a) What is the maximum number of items that will be in the new store's inventory?

b) What is the location of the store?

c) How many items will be in the initial inventory?

d) Repeatedly present a menu to allow the user to select the generic category of an item (an integer) and then request the information for that item until all items have been entered.

e) Repeatedly present the user with the following menu until a "3" is entered:

Enter 1 to output all the information for all of the items in the inventory

Enter 2 to output all the information for all of the items in a particular generic category

Enter 3 to quit the program.

Enrichment

1. Why does Java not allow multiple inheritance? How is the diamond problem a consequence of multiple inheritance?

2. How can most of the features of multiple inheritance be simulated in Java?

3. Oracle's Javadoc materials:
https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

CHAPTER 9

RECURSION

In this chapter

In this chapter we introduce recursion, a very powerful tool used in problem solving in which a problem's solution is expressed in terms of a simpler version of itself. The implementation of the recursive solution results in a method that invokes itself. We will examine the execution path of a recursive method, explore a methodology for discovering and implementing recursive algorithms, and practice this methodology on a set of progressively more difficult problems.

Although recursion can provide succinct and eloquent solutions to many problems, such as the Towers of Hanoi problem, the drawing of fractals, and the solution to many puzzles, the implementation of some recursive solutions can produce runtime problems. They can require large amounts of RAM memory and can result in unacceptably long execution times. We will identify the characteristics of recursion that cause these problems and learn a technique called dynamic programming used to solve one cause of unacceptably long execution times.

After successfully completing this chapter you should:

- Understand recursion and a recursive method's execution path
- Be able to design and implement a recursive solution to a problem by discovering its base case, a reduced solution, and a general solution
- Understand why recursive solutions require more time and storage than their iterative counterparts
- Know when to use a recursive solution and when an iterative solution would be more practical or efficient
- Understand how dynamic programming techniques can improve the efficiency of a recursive algorithm

9.1 WHAT IS RECURSION?

In general, recursion is defining something in simpler terms of itself, a property often referred to as self-similarity. In computer science, recursion is a technique used in the coding of methods and formulating algorithms. Usually, these methods or algorithms refer back to or are applied to themselves. When this technique is used to code a method, we say that the method is *recursive*. Before the execution of a recursive method ends, it either invokes itself or another method whose execution eventually leads to an invocation of the recursive method. Figure 9.1 illustrates both of these forms of recursion.

```
public int aRecusiveMethod(int p)
{
   //code of the beginning
   //of the method

   int value = aRecusiveMethod(21)

   //remaining code of the method
}
```

A Method Reinvoked by Itself

```
public void methodA()

  //beginning code
  //of the methodA

   methodB();

   //remaining code
   //of the method
}
```

A Method Eventually Reinvoked by Another Method

```
public void methodB()

  //beginning code
  //of the methodB

   methodC();

   //remaining code
   //of the method
}
```

```
public void methodC()
{
  //beginning code
  //of the methodC

   methodA();

   //remaining code
   //of the method
}
```

A Reinvocation of `methodA`

Figure 9.1
The two forms of recursive methods.

In mathematics, recursion is often used to define functions and series that can also be defined without using recursion. For example, a non-recursive definition of ten factorial (10!) is:

$$10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1$$

This non-recursive definition of 10! can be generalized to a non-recursive definition of **n!** for any positive value of n.

Definition

The non-recursive definition of **n!** for values of $n \geq 0$

$\mathbf{n!} \equiv n * (n - 1) * (n - 2) * (n – 3) * (n – 4)* ....* 3 * 2 * 1$ *and* $0! \equiv 1$

Alternately, the definition of 10! can be more succinctly stated recursively as

$$10! \equiv 10 * (10 - 1)! \quad \textit{and} \quad 0! \equiv 1$$

This recursive definition can be generalized to a recursive definition of **n!** for any positive value of n.

Definition

The recursive definition of **n!**

n! $\equiv n * (n - 1)!$ *and* $0! \equiv 1$

The equivalence of the recursive and non-recursive definitions can be easily understood by examining the non-recursive definition of 10! and 9!

$$10! = 10 * \underbrace{9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1}_{(10\,-\,1)! \,=\, 9!} \qquad 9! = 9 * \underbrace{8 * 7 * 6 * 5 * 4 * 3 * 2 * 1}_{(9\,-\,1)!}$$

The last nine terms in the equation for 10! are contained in the right side of the equation for 9!, so 10! can certainly be expressed as 10 * 9!, which is the recursive definition of 10!. Having been shown this example, most of us would accept the fact that 9! can be used to calculate 10!, that is:

$$10! = 10 * 9!$$

This is the recursive way of calculating 10! which we now understand because we have realized that 9! = 9* 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1. In effect, we have used the non-recursive definition of 9! to understand the recursive definition of 10!.

A recursive definition typically appears to define an entity in simpler terms of itself. The two uses of the factorial operator in the recursive definition of n! could be considered an example of this because the definition states that for positive values of n,

"n *factorial* is equal to n times n minus one *factorial.*"

What makes this part of the recursive definition acceptable is that, although the word "factorial" is used in two phrases that make up the sentence ("n *factorial*" and "n minus 1 *factorial*"), the two phrases are different. In addition, the recursive definition of n! contains another crucial part: zero factorial is defined as one (0! ≡ 1).

The reason this second part of the definition is crucial to the validity of a recursive definition is that the recursive definition relies on being able to calculate (n – 1)! in order to calculate n!. Knowing that the value of zero factorial is defined as one gives us the ability to calculate (n – 1)!. This fact may come as a surprise, but it becomes more obvious after a close examination of Figure 9.2, in which the recursive definition of n! is used in a progressive way to calculate (4)!. The right side of line 6 completes the calculation by substituting the value one (1), which is defined to be the value of zero factorial (0!)

The recursive definition of n factorial: $n! \equiv n * (n - 1)!$ *and* $0! \equiv 1$.

```
1    4! = 4 * (4 - 1)!
2       = 4 * 3!
3       = 4 * (3 * 2!)
4       = 4 * (3 * (2 * 1!))
5       = 4 * (3 * (2 * (1 * 0!)))
6       = 4 * (3 * (2 * (1 * 1)))
7       = 24
```

Figure 9.2
Calculating 4! using the recursive definition of n factorial.

The progression from lines 2 to 5 of Figure 9.2 is analogous to the progression of four phone calls shown at the top of Figure 9.3 initiated by you after your friend Evie asks you the value of four factorial. Remembering the recursive definition of n factorial, you know 4! is 4 * 3!. Because you don't know the value of 3!, you call your friend Logan and ask him the value of 3!. He realizes it is 3 * 2! and calls his friend Skyler to ask her the value of 2!, who calls Ryan to ask him the value of 1!. Finally, Ryan calls Breanne to ask her the value of 0!,

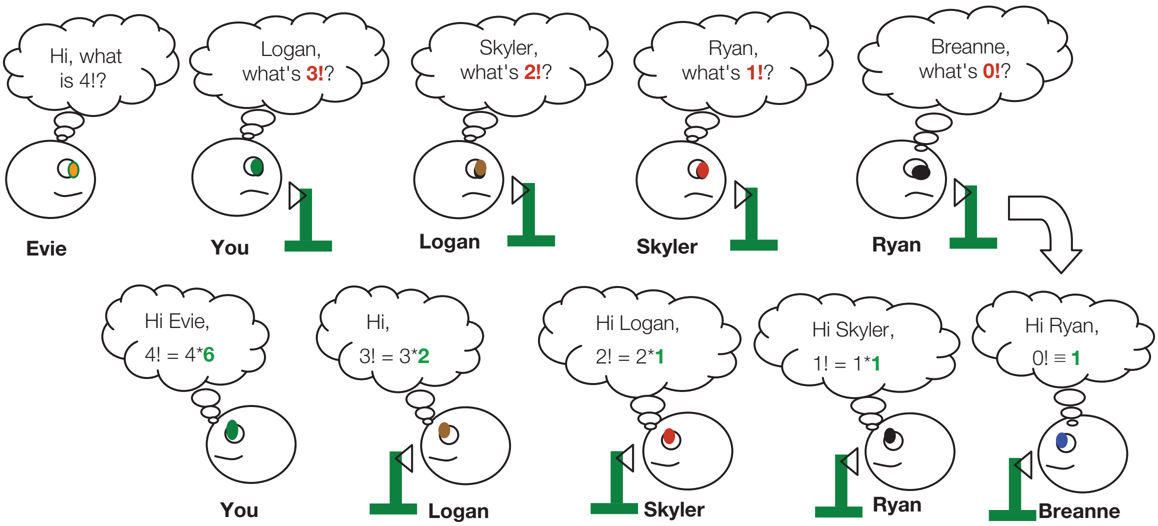


Figure 9.3
A progression of five phone calls to determine the value of four factorial.

This leads to the end of the sequence of four phone calls as depicted at the bottom-right of Figure 9.3. In response to Ryan's call, Breanne examines the definition of 0! and tells Ryan the value of 0! is *defined* as 1 and hangs up. Ryan multiplies 1 by the value of 0! that Breanne told him, 1, tells Skyler the value of 1! is 1, and hangs up his phone. This scenario continues with Skyler telling Logan the value of 2! is 2, and Logan telling you the value of 3! is 6. Finally, you multiply 4 by the value of 3! that Logan told you, 6, and you tell Evie the value of 4! is 24.

Recursion is not only used to more succinctly define formulas and series in mathematics, but it is also used to more succinctly define algorithms in computer science. It is for this reason that programming languages permit the two types of recursive invocations illustrated in Figure 9.1. If we were going to write a method that calculated n!, and the recursive definition of n! was used as the method's algorithm, then, when it was used to calculate 4!, the sequence of four phone calls illustrated in Figure 9.3 would be the method invoking itself four times. Evie asking you the value of 4! would be the initial invocation of the method.

Not all algorithms can be expressed recursively, but the effort expended in implementing those that can be is significantly less than implementing a non-recursive version of the algorithm. For this reason, an understanding of how recursive methods execute, how to formulate and implement a recursive algorithm, and some problems associated with the use of recursion is an essential tool to have in our programming toolbox.

9.2 UNDERSTANDING A RECURSIVE METHOD'S EXECUTION PATH

A recursive method is invoked, like any other method, by coding its name followed by an argument list used to pass values to the method's parameters. If the method returns a value, the value is either assigned to a variable for use in subsequent statements or used in the statement that contains the invocation of the method. Looking at the invocation statement, there is no way to determine if the method being invoked is recursive or not. Below is an invocation of a recursive method named `fact` that returns 4!, which could also be the invocation of a method with the same signature that is non-recursive:

```
long fourFactorial = fact(4);
```

The recursive version of this factorial method would differ from the non-recursive version within the implementation of the method. The recursive version implements the recursive definition of n factorial using the coding model shown at the top of Figure 9.1. Figure 9.4 shows the code of the method `fact` that calculates the value of n! recursively.

```
public long fact(int n)
{
  long nMinus1Factorial, nFact;
  if(n == 0) //return the definition of 0!
  {
    return 1;
  }
  else //calculate (n-1)! and then n!
  {
    nMinus1Factorial = fact(n-1); //fact invokes itself here
    nFact = n * nMinus1Factorial;
    return nFact;
  }
}
```

Figure 9.4
A recursive method that calculates n!.

If the Boolean condition on line 4 of Figure 9.4 is `true`, line 6 returns 1, the value of 0! specified in the definition of n!. Otherwise, line 10 of the method invokes itself. Consistent with the first part of recursive definition of n!, it passes `fact` the value of n-1, and the method calculates and returns the value of (n-1)!. Then line 11 uses the returned value to calculate n factorial, which is returned on line 12.

Although the calculations performed by lines 10 and 11 are coded sequentially, they do not execute sequentially because line 10 of the initial invocation of the method initiates the first of the progression of four phone calls depicted at the top of Figure 9.3. When calculating 4!, the first of these phone calls (or, more accurately, recursive *invocation*s) is passed the value 3 (i.e., 4 - 1).

The three subsequent recursive invocations of the method by line 10 pass the values 2, 1, and finally 0 to the method. The line-by-line execution sequence of the method to reach this point, beginning with the initial invocation of the method, is given below and is represented by the red arrows in Figure 9.5.

Lines 1, 2, 3, 4, 8, 9, 10 //which passes the *first* recursive invocation the value 3
Lines 1, 2, 3, 4, 8, 9, 10 //which passes the *second* recursive invocation the value 2
Lines 1, 2, 3, 4, 8, 9, 10 //which passes the *third* recursive invocation the value 1
Lines 1, 2, 3, 4, 8, 9, 10 //which passes the *fourth* recursive invocation the value 0

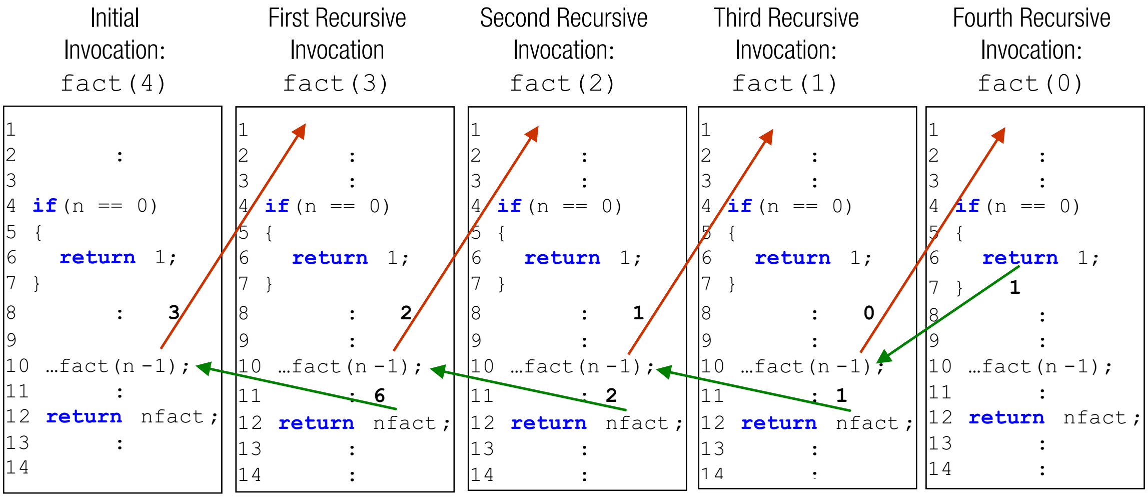


Figure 9.5
The execution sequence of the invocations of the method **`fact`** to calculate 4!

Because the fourth recursive invocation of the method is passed the value zero, its execution sequence is lines 1, 2, 3, 4, 5, and 6 because the Boolean condition on line 4 for this execution of the method evaluates to `true`. As a result, line 6 returns the value 1 (0!) to line 10 of the third recursive invocation (as indicated by the green arrow on the right side of Figure 9.5). Then lines 11 and 12 of the third, second, and first recursive invocations execute, returning the values 1, 2, and 6 (as indicated by the other three green arrows in Figure 9.5). Finally, lines 11 and 12 of the initial invocation of the method execute and return 24 = 4 * 6.

Two important observations to make from an examination of Figure 9.5 are that:

1. because of the recursive invocations spawned by line 10 of the initial invocation of the method, a considerable amount of time is required to complete the execution of this line of code
2. the recursive invocations of the method would have continued had it not been for the fact that the fourth recursive invocation was passed the value zero

Armed with an understanding of the execution path of recursive methods, we will discuss the techniques used to discover recursive algorithms and the nuances of implementing them.

9.3 FORMULATING AND IMPLEMENTING RECURSIVE ALGORITHMS

Most people do not possess an innate ability to think recursively. When shown the non-recursive and recursive definitions of n factorial, most of us would say that we are able to understand the non-recursive definition, but we are confused by its recursive counterpart. Therefore, it should come as no surprise that most of us have trouble recognizing that there is a recursive solution to a problem and have even more trouble trying to discover the algorithm even after we are told that one does exist.

Fortunately, we can learn to think recursively because the discovery of recursive algorithms can be methodized, and once discovered, many recursive algorithms are implemented in a very similar way. After gaining a basic understanding of the methodized process, lots of practice facilitates the learning curve.

9.3.1 The Base Case, Reduced Problem, and General Solution

The recursive-algorithm discovery process is broken into four discovery steps:

1. identify the *base case* or cases
2. identify the *reduced problem* or problems
3. identify the *general solution*
4. combine the base case, reduced problem, and general solution into a recursive algorithm

We have already used a base case, reduced problem, and general solution in the program presented in Figure 9.4 without identifying them by name. The base case, reduced problem, and general solution of the recursive algorithm for n factorial implemented in Figure 9.4 are identified in Table 9.1.

Prior to coding the method shown in Figure 9.4, we did not discover these three parts of the recursive algorithm for n!. They were discovered by the person that originally formulated the recursive definition of n!, who was probably born with the ability to think recursively. We simply discussed a method that implemented this definition of n! to gain insights into the execution path of a recursive method. This was a useful exercise because an understanding of the method's execution path, and an awareness of these three parts of the recursive definition of n!, do facilitate an understanding of the discovery process.

Table 9.1
The Base Case, Reduced Problem, and General Solution for n!

Base Case	Reduced Problem	General Solution
$0! \equiv 1$	$(n - 1)!$	$n * (n - 1)!$

Discovering the Base Case

The recursive-algorithm discovery process begins with the identification of the problem's base case. To discover the base case, we search for a particular instance of the problem whose solution is known. For example if we were trying to discover the recursive algorithm for n factorial, we would ask ourselves if there is a particular value of n: 0, or 1, or 2, or 3, or 4, etc., whose factorial value is known. Most people know that 0! is defined as 1, which is the base case for n factorial. The base case for another problem, x^n, is also a definition: x^0, which is also defined as 1. For some problems, the base case is not a definition but a trivial case of the problem that most of us know, such as $1 * 1 = 1$. Another example of a trivial base case is associated with the problem of outputting a string of n characters in reverse order. In this case, when n is one, we simply output the string.

When the problem involves an integer n, often (as we have seen) the base case is the solution to the problem when $n = 0$ or, for some problems, when $n = 1$. The determination of the base case is a crucial first step in the discovery process because it is used to discover the reduced problem, which is then used to discover the general solution. In addition, when we implement the algorithm, as shown on line 4 of Figure 9.4 and on the far right side of Figure 9.5, it is the base case that halts the progression of recursive invocations and is sometimes referred to as the *stopping condition*. Some problems, as we will see, contain multiple base cases.

Discovering the Reduced Problem

Identifying the reduced problem is often the most difficult part of the four-step discovery process. Usually, it can be identified by considering the following three criteria. The reduced problem:

1. is a problem similar to the original problem
2. is usually between the original problem and the base case and is usually closer to the original problem than it is to the base case
3. becomes the base case for all versions of the original problem when *progressively reduced*

For example, the reduced problem for n! is $(n - 1)!$. Clearly, this is similar to the original problem in that it also involves n and uses the factorial operator. It is between the original problem and the base case (0!), and for most values of n ($n > 2$), it is closer to the original problem than it is to the base case. This would be an acceptable reduced problem if progressive reductions caused it to become the base case for all values of n.

By progressive reductions, we mean that we repeatedly apply the relationship between the candidate reduced problem and the original problem to the reduced problem to produce new versions of the problem. For n!, the relationship is that the number used in the reduced problem (n – 1) is one less than the number used in the original problem (n). Applying this relationship to the reduced problem (n – 1)!, the new version of the problem becomes ((n – 1) – 1)! = (n – 2)! on the first reduction. Subsequent new versions of the problem are ((n – 2) – 1)! = (n – 3)!, (n – 4)!, (n – 5)!, etc. Because this progressive reduction of (n – 1)! for any positive value of n will eventually become the base case, (n – 1)! is an acceptable reduced problem for n!.

An example of an unacceptable reduced problem for n! is (n – 2)!, although it satisfies most of our criteria. It is a problem similar to the original problem, it is between the base case (0!) and the original problem, and it is also closer to the original problem than it is to the base case. What makes it unacceptable is that progressive reductions, for *odd* values of n, do not result in the base case. For example, for n = 7 the reduced problem would be (7 – 2)! = 5!, and its progressive reductions would be:

$$(5 - 2)! = 3!, \quad (3 - 2)! = 1!, \quad (1 - 2)! = -1!.$$

The reductions skip over the base case, zero factorial. They never become the base case for odd values of n. Another unacceptable candidate reduced problem is (n + 1)!. For n = 7, the reduced problem would be (7 + 1)! = 8!, and its progressive reductions would be 8!, 9!, 10!, 11!, etc. The result would be an infinite series that never becomes the base case.

Some problems, as we will see, contain multiple reduced problems.

Discovering the General Solution

The general solution is the solution to the original problem. To discover the general solution, we think of a way to solve the original problem assuming that we have already found a valid reduced problem. That is, we think of a way of use the solution to the reduced problem in the solution of the original problem.

In the case of n!, this translates into the question of how (n – 1)! can be used to calculate n factorial. The answer is multiplying (n – 1)! by n, so the general solution for n! is n × (n – 1)!

The first three steps of the recursive-algorithm discovery process are summarized in Table 9.2.

9.3.2 Implementing Recursive Algorithms

Once we have discovered the base case, reduced problem, and general solution for a particular problem, they are combined into a recursive algorithm. Finding the correct combination usually involves some creativity, which comes naturally to some people while the rest of us have to develop this creative ability through practice.

For many problems, the base case, reduced problem, and general solution are combined in a similar way. For a subset of these problems, they are combined in an identical way, which is depicted in the flow chart shown in Figure 9.6. By working with this subset of problems, we will begin to develop the ability to think recursively. Continued practice with the super set of problems will further develop our skills.

Table 9.2
The Recursive Algorithm Discovery Process

Step	Process	Comments
1: Base Case Discovery	Search for a particular instance of the problem whose solution is known, because it is a defined solution or it is a trivial solution that is known to most people.	The base case for n! is 0! ≡ 1 If the problem involves an integer n, the base is often when n = 0 or n = 1.
2: Reduced Problem Discovery	Search for a version of the problem that: • is similar to the original problem • usually is between the original problem and the base case *and* closer to the original problem • when progressively reduced becomes the base case for *all versions* of the original problem	The reduced problem for n! is (n – 1)! n! ≤ (n – 1)! ≤ 0! The progressive reductions of (n – 1)! are: (n – 1)!, (n – 2)!, (n – 3)!, etc., which becomes the base case, 0!, for all values of n.
3: General Solution Discovery	Think of a way to solve the original problem, assuming that we have already found a solution to the reduced problem.	The reduced problem is used in the general solution, e.g., n! = n * (n – 1)!

The algorithm shown in Figure 9.6 is most easily understood by comparing it to a particular implementation of it: the code of the method to recursively compute n! presented in Figure 9.4. The problem of calculating n! is one of the problems in the subset of problems whose three components (base case, reduced problem, and general solution) can be combined exactly as shown in Figure 9.6.

Figure 9.7 represents the code of this method with comments added to it to aid in an understanding of the manner in which the algorithm combines its three components. These comments refer to the blue-font numbers shown on the left side of each of the flow-chart symbols shown in Figure 9.6.

The `if` statement coded on line 4 of Figure 9.7 represents the question in symbol 2 of the flowchart, which is used to decide if the problem to be solved is the base case. The `return` statement (line 6) inside the `if` statement's code block represents flowchart symbols 3 and 4 that terminate the algorithm after returning the base case solution.

Symbol 5 of the flowchart is typically the most confusing part of the algorithm because it is the most confusing part of recursion. It calculates the solution to the reduced problem using this same algorithm. It is the recursive part of the algorithm. The right side of line 10 implements symbol 5 by re-invoking the method and passing it the reduced problem. As we have learned, this symbol of the flowchart can occupy a significant portion of the algorithm's execution time. Every time it is entered, it spawns another execution of the algorithm, beginning at symbol 1, to calculate the solution to a new (reduced) factorial problem.

After flowchart symbol 5 calculates the solution to the reduced problem, symbol 6 uses it to calculate the general solution to the original problem. Line 11 of Figure 9.7 implements flowchart symbol 6. After symbol 7 returns the problem's solution (line 12 of Figure 9.7), symbol 8 ends the algorithm.

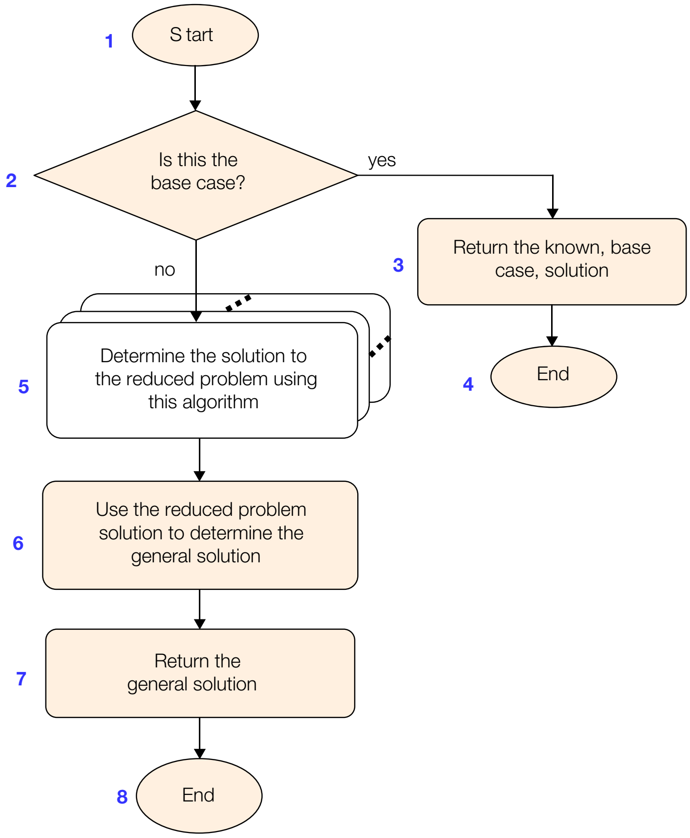


Figure 9.6
A template for some recursive algorithms.

```
1   public long fact(int n)
2   {
3     long nMinus1Factorial
4     if(n == 0) //Flow chart symbol 2
5     {
6       return 1; //Flow chart symbols 3 and 4
7     }
8     else
9     {
10      nMinus1Factorial = fact(n-1); //Flow chart symbol 5
11      nFact - n * nMinus1Factorial; //Flow chart symbol 6
12      return nFact; //Flow chart symbols 7 and 8
13    }
14  }
```

Figure 9.7
A recursive method that calculates n! commented to correlate to the algorithm presented in Figure 9.6.

9.3.3 Practice Problems

Now that we have acquired a basic understanding of recursion, the best way to extend our ability to think recursively is to apply our newly acquired skills to a set of problems that can be solved recursively. The problems presented in Table 9.3 are tabulated in an order intended to extend these skills in an incremental manner. As we move from the problems at the top of the table to those at the bottom of the table, their recursive solutions become increasingly dissimilar to the recursive solution n!. For this reason, which is consistent with the adage that "practice makes perfect," it is best to work our way through all of the problems in the order in which they are tabulated. Table 9.4 (included in the Chapter Summary) presents the base case, reduced problem, and general solution for each of the problems presented in Table 9.3

Table 9.3
Several Problems That Have Recursive Solutions in Difficulty Order

Problem	Base Case	Reduced Problem	General Solution
Factorial of a positive integer, n!	0!≡ 1	(n-1)!	n * (n-1)!
A number x raised to a positive integer, x^n			
Sum of the integers from 1 to n			
Product of two positive integers, m × n			
Output an n character string, s, in reverse order			
Problems with Multiple Base Cases and Reduced Problems			
Generate the n^{th} term of the Fibonacci Sequence, fn			
Find the greatest common divisor of two positive integers m and n, GCD(m, n), for m > n			
A Problem with Multiple Base Cases			
Search a sorted list of items for I			
A Problem that Uses the Reduced Problem Twice			
Towers of Hanoi: Move n rings from tower A to tower B using tower C			

As noted in the table, some problems have multiple base cases and reduced problems, and one problem uses its reduced problem twice in its general solution. As we have learned, the starting point is the determination of a problem's base case, then its reduced problem, and then its general solution. They should be entered into a copy of the last three columns of the table below the column entries for n!, and in most cases, they can be combined into a recursive solution using the algorithm depicted in Figure 9.6.

The correctness of the discovered base case, reduced problem, and general solutions should be verified by comparing them to the entries in Table 9.4 in the Chapter Summary. Implementing each problem's recursive solution before moving on to the next problem is highly recommended because it augments the learning process.

It is not unusual in the beginning, or at some other point in this learning process, not to be able to discover one or more of the three components of the recursive solution (i.e., its base case, reduced problem, or general solution). When this is the case, it is still useful to implement the recursive algorithm using the component(s) presented in Table 9.4. Consistent with this idea, many students begin by skipping the discovery process and simply implement a recursive method to calculate x^n using the components given in the second row of Table 9.4 and the flow chart presented in Figure 9.6.

A description of the last problem in Table 9.3, The Towers of Hanoi, is given in the beginning of the next section.

9.4 A RECURSION CASE STUDY: THE TOWERS OF HANOI

The recursive solution to the Towers of Hanoi problem is a good example of the simplicity and power of recursive algorithms. Its recursive solution can be implemented as seven lines of code. Applying the techniques we have learned to produce a recursive solution, this problem will serve as a good capstone to our discussion of how to formulate and implement recursive algorithms. Not only will this reinforce our knowledge of the discovery process, but the implementation of the problem's recursive algorithm will introduce an often-used nuance into the parameter list of a recursive method that is not used in the solutions of the other problems presented in Tables 9.3 and 9.4.

Statement of the Problem

The problem was conceived by the French mathematician Eduardo Lucas in 1883. It involves three towers and a set of n disks or rings, each with different diameters. The original legend claimed that the Brahmins of an ancient Indian temple were charged with moving 64 golden disks, and when the last one was in its final place, the world would come to an end.[1] As we will see, there is no need to worry about this because the disks had to be moved in a specified order, and if one disk were move per second it would take 585 billion years (2^{64} - 1 seconds) to relocate the disks.

The left side and right sides of Figure 9.8 illustrate the problem's starting and ending points for four rings (n = 4). The rings are initially stacked in decreasing order by size on one of the towers, which we will refer to as the starting tower. The left side of Figure 9.8 shows four rings stacked on a starting tower named A. The right side of the figure shows the four rings relocated to the tower named B, which we will refer to as the destination tower. We will refer to the third tower shown in the figure, whose name is C, as the extra tower.

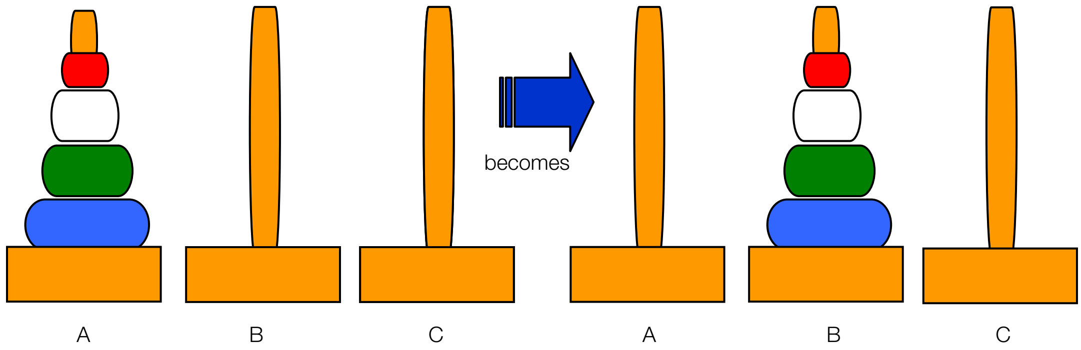


Figure 9.8
The Towers of Hanoi problem's starting and ending points using four rings.

The solution to the problem is a specification of the order in which to move the rings that will relocate all of them from a designated starting tower (tower A) to a designated destination tower (tower B) without violating the following two conditions:

1. Only one ring (a top ring) can be moved at a time, and it must be placed on a tower before another ring is moved.
2. A larger ring cannot be placed on top of a smaller ring.

These conditions imply that when the problem is solved, the rings will be stacked on the designated destination tower in decreasing size order, as shown on the right side of Figure 9.8. The problem solutions (i.e., the order in which the rings are moved to relocate them from tower A to tower B) for two, three, and four rings are given in Figure 9.9. Figure 9.10 depicts the three-ring solution.

Notice that there is a pattern to predicting the number of moves that will be needed for n rings. For two rings, there are three moves, for three rings, there are seven moves, and for four rings, there are fifteen moves. In general, the minimum number of moves for n rings will be 2^n -1. This is why moving the legendary 64 golden disks would require 2^{64} -1 moves and extraordinarily long time.

The Base Case

We have learned that the formulation of a recursive solution to a problem begins with a search for its base case: a known, defined, or trivial solution to the problem. We have also learned that when the problem involves n items, a good place to begin the search is when n = 0 or n = 1. This problem involves n rings, so we will begin by considering the case when there are zero rings. This is a legitimate base case for this problem because the trivial solution would be to do nothing. An alternate base case is when n = 1 because if we were asked to move one ring from tower A to tower B, we would simply state "move the top ring on tower A to tower B." Both of these are trivial solutions because most people would know them, and they do not require a consideration of the two conditions of the Towers of Hanoi problem. Either of these base cases can be used in the recursive solution to the problem. The n = 1 base case is the one that appears in Table 9.4.

Two-Ring Solution	**Four-Ring Solution**
1- move one ring from tower A to tower C	1- move one ring from tower A to tower C
2- move one ring from tower A to tower B	2- move one ring from tower A to tower B
3- move one ring from tower C to tower B	3- move one ring from tower C to tower B
	4- move one ring from tower A to tower C
Three-Ring Solution	5- move one ring from tower B to tower A
1- move one ring from tower A to tower B	6- move one ring from tower B to tower C
2- move one ring from tower A to tower C	7- move one ring from tower A to tower C
3- move one ring from tower B to tower C	8- move one ring from tower A to tower B
4- move one ring from tower A to tower B	9- move one ring from tower C to tower B
5- move one ring from tower C to tower A	10- move one ring from tower C to tower A
6- move one ring from tower C to tower B	11- move one ring from tower B to tower A
7- move one ring from tower A to tower B	12- move one ring from tower C to tower B
	13- move one ring from tower A to tower C
	14- move one ring from tower A to tower B
	15- move one ring from tower C to tower B

Figure 9.9
Three Towers of Hanoi problem solutions.

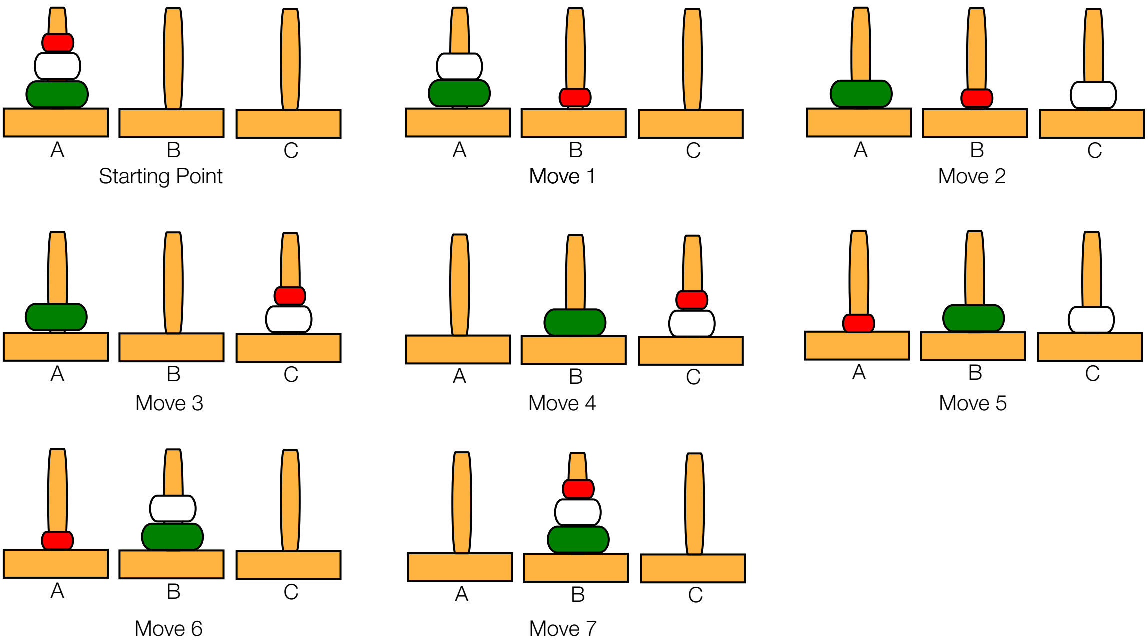


Figure 9.10
The solution to the three-ring Towers of Hanoi problem.

NOTE

Towers of Hanoi base case:
$n = 1$: Move 1 ring from one tower to another tower

Reduced Problem

To discover the reduced problem, we can use a new tool often employed when the problem involves n items. We try to solve very simple versions of the problem, not quite as trivial as the base case, but close to it. Then, we examine the solutions looking for similarities and try to generalize the similarities into a reduced problem. For example, suppose we were able to produce the two- and three-ring solutions shown in Figure 9.9, which many non-recursive thinking people could probably do. If we are visual learners, it helps to sketch out each step of the solutions, as has been done in Figure 9.10 for the three-ring solution.

Examining Figure 9.10 and the two-ring solution in Figure 9.9, we notice that just before the half-way point of both solutions (move 4 of Figure 9.10 and move 2 of the two-ring solution), all of the rings except for one have been moved to tower C. Examining the moves made after the mid-point of both solutions (moves 5–7 in Figure 9.10 and move 3 of the two-ring solution in Figure 9.9), we realize that these also move all of the rings except for one.

Because the ability to move all but one ring from one tower to another tower is common to both the two- and three-ring solutions, perhaps this could be the reduced problem. When generalized, it becomes: move n-1 rings from one tower to another tower. This generalization does satisfy the three criteria for a reduced problem restated below and is the one that appears in Table 9.4:

1. It is similar to the original problem: move n rings from one tower to another tower.
2. It is closer to the original problem than either of the problem's base cases (n = 0 and n = 1), and it is between the original problem and these base cases.
3. When progressively reduced, it does eventually become one of the base cases: move one or move zero rings.

Another way of discovering this reduced solution would be to compare the illustration of the starting point in Figure 9.10 to the illustration of move 3 in that figure and to compare the illustrations of move 4 to move 7. From these comparisons, we would be likely to observe that the problem could be solved if we knew how to move n-1 rings, and then conclude that we need to make a call to a friend who knows how to move n-1 rings from one tower to another tower.

NOTE *Towers of Hanoi reduced problem:*
Move n-1 rings from one tower to another tower.

The General Solution

To discover the general solution, we ask ourselves how we can use the reduced problem to solve the original problem. In the case of the Towers of Hanoi, this question becomes: "how can we use the ability to move *n-1* rings to solve the problem of moving n rings from one tower to another tower?" The answer can often be found by examining the solutions developed to the very simple versions of the problem previously used to determine the reduced problem.

The three move sequences on both sides of move 4 in Figure 9.10, which shows the n = 3 solution, represent a movement of 3-1 rings from one tower to another. This means that if move

4 of Figure 9.10 was preceded and followed with a version of the reduced problem, it would be the general solution of the n = 3 ring problem:

1. use the reduced problem to move 2 rings from tower A to C (moves 1–3)
2. move 4 (move 1 ring from tower A to tower B)
3. use the reduced problem to move 2 rings from tower C to B (moves 5–7)

Once we have found a way to use the reduced problem to solve one of the simpler problems used in the discovery of the reduced problem (e.g., the 3 ring problem), its use is extrapolated to the n ring general solution. This is the general solution presented in Table 9.4.

NOTE

Towers of Hanoi general solution:
1. *use the reduced problem to move n-1 rings from tower A to C*
2. *move 1 ring from tower A to tower B*
3. *use the reduced problem to move n-1 rings from tower C to B*

Implementation

The base case, reduced problem, and general solution of this problem can be combined into a recursive algorithm using the flow chart shown in Figure 9.6. The application presented in Figure 9.11 implements this algorithm in a method named `hanoi` (lines 8–24). The signature of the method contains four parameters: the number of rings, followed by the name of the starting tower, the name of the destination tower, and the name of the third tower. It is invoked on line 5 to output the moves required to transfer four rings from tower A to tower B. The output produced by the program is shown in Figure 9.12.

As the names of the last three parameters in the signatures of the `hanoi` method on lines 8 and 9 indicate, the method uses the tower name passed to its parameter (`fromTower`) as the starting tower, and the tower names passed to the method's second and third parameters (`toTower and thirdTower`) are used as the destination tower and the extra tower, respectively. As a result, when the method is invoked recursively on line 20 of the general solution to move all but the bottom ring from the starting tower to the extra tower, the second argument used in the invocation is the parameter `thirdTower`, and the last argument is the parameter `toTower`. This effectively swaps roles of the extra tower and the destination tower during this invocation of the method, and n-1 rings are relocated to the extra tower.

Similarly, when the method is invoked on line 23 of the general solution to move the n-1 rings placed on the extra tower from the extra tower to the destination tower, the first argument used in the invocation is the method's parameter `thirdTower`, and the last argument is the method's parameter `fromTower`. This effectively swaps roles of the extra tower and the starting tower during this invocation of the method.

When the method `hanoi` is invoked, and line 12 detects the base case (the value passed to `nRings` is 1), the output produced by lines 14–15 includes the names of the towers passed to the method's first and second parameters `fromTower` and `toTower`. This ensures that when the invocations on lines 20 and 23 degenerate to the base case, the values of the first and second arguments passed to the method will be included in the base case's output (lines 14–15).

```
public class TowersOfHanoi
{
  public static void main(String[] args)
  {
    hanoi(4, "A", "B", "C"); //output the solution for four rings
  }

  public static void hanoi(int nRings, String fromTower,
                           String toTower, String thirdTower);
  {

    if(nRings == 1) //base case
    {
      System.out.println("move one ring from tower " + fromTower +
                         " to tower " + toTower);
      return;
    }

    //general solution
    hanoi(nRings-1, fromTower, thirdTower, toTower); //reduced problem
    System.out.println("move one ring from tower " + fromTower +
                       " to tower " + toTower);
    hanoi(nRings-1, thirdTower, toTower, fromTower); //reduced problem
  }
}
```

Figure 9.11
The application **TowersOfHanoi**.

```
move one ring from tower A to tower C
move one ring from tower A to tower B
move one ring from tower C to tower B
move one ring from tower A to tower C
move one ring from tower B to tower A
move one ring from tower B to tower C
move one ring from tower A to tower C
move one ring from tower A to tower B
move one ring from tower C to tower B
move one ring from tower C to tower A
move one ring from tower B to tower A
move one ring from tower C to tower B
move one ring from tower A to tower C
move one ring from tower A to tower B
move one ring from tower C to tower B
```

Figure 9.12
The output produced by the application **TowersOfHanoi**.

9.5 PROBLEMS WITH RECURSION

Because only a small percentage of the population has an innate ability to think recursively, most of us need to be trained in how to discover and implement recursive algorithms. To a certain extent, the discovery and implementation process can be methodized, but a good deal of effort is necessary to become a good recursive programmer.

Another problem with recursion is that applications that use recursive algorithms tend to run more slowly. If two versions of the same application were developed, one that used a recursive algorithm and one that used a non-recursive algorithm, the non-recursive version would typically run faster. The difference in speed is due to the manner in which modern computer systems transfer execution to, and return from, an invoked method and the larger number of method invocations that recursive algorithms typically perform by repeatedly invoking themselves.

Every time a method is invoked, whether or not it is recursive, the runtime environment has to suspend the execution of the program to perform tasks associated with the invocation. Typically, these tasks include allocating the memory for the invoked method's parameters and local variables and transferring the value of the arguments into these parameters. Not only does this take time, but each method invocation requires additional RAM memory for the storage of the method's parameters and local variables.

Storage must also be allocated to save the contents of the CPU registers and the invoking method's return address to complete the method's execution. Java stores this information in an area of storage called the *run-time stack*. After each invocation completes its execution, the invoking method cannot continue its execution until the information is retrieved from the run-time stack and stored in the CPU's registers.

As shown in Figure 9.5, the recursive method to calculate n! invokes itself four times to calculate 4!; to compute 40! requires 40 invocations, and n! requires n invocations. A non-recursive, or *iterative*, version of the method is given in Figure 9.13. It does not issue any method invocations during its execution regardless of the value of n. Its speed advantage over the recursive version increases with increasing values of n. In addition, it does not require storage allocated for the parameters, local variables, CPU register values and return addresses associated with the additional invocations of the recursive version of the method.

```
public static long factIterative(int n)
{
  long nFact = 1;
  for(int i = n; i >= 1; i--)
  {
    nFact = nFact * i;
  }
  return nFact;
}
```

Figure 9.13
An iterative method that calculates n!.

The number of invocations issued by some recursive algorithms can make the recursive solution incalculable from a time viewpoint. The implementation of the recursive definition of the **Fibonacci sequence** is one such example.

Definition

The recursive definition of the terms of the **Fibonacci sequence**:

$f_1 \equiv 1$ and $f_2 \equiv 1$;

for $n \geq 2$: $f_n = f_{n-1} + f_{n-2}$

Consistent with this definition, the first eight terms of the sequence are 1, 1, 2, 3, 5, 8, 13, and 21. This recursive definition has two base cases, $f_1 \equiv 1$ and $f_2 \equiv 1$, two reduced problems, f_{n-1} and f_{n-2}, and its general solution is $f_n = f_{n-1} + f_{n-2}$. While this may look innocent, when these base cases, reduced problems, and the general solution are combined as shown in Figure 9.6 and implemented into a recursive method, the number of recursive invocations made to calculate the 40th term in the sequence, f40, is 204,668,309. The reason there are this many invocations is illustrated in Figure 9.14, where the arrows in the figure should be interpreted as the method invoking itself to calculate a term of the series. Referring to the top of the figure, to calculate f40, the method is invoked to calculate f39 and f38. To calculate f39, the method is invoked to calculate f38 and f37. As shown in the remainder of the figure, during the recursive definition of f40, f39 is calculated once, but f38 is calculated twice, f37 three times, f36 five times, f35 eight times, etc.

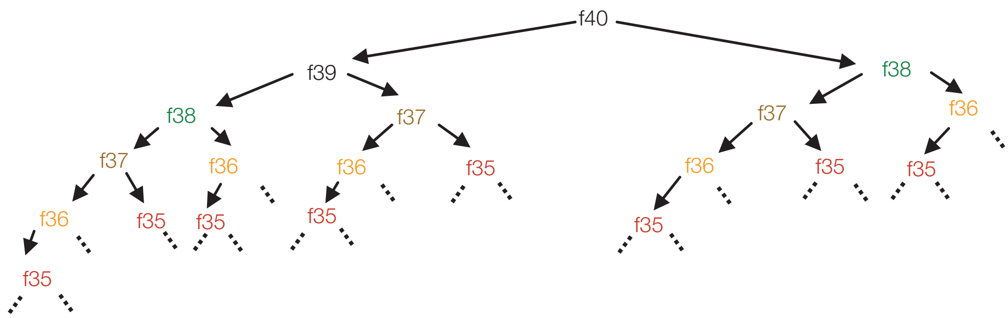


Figure 9.14
Some of the 204,668,309 invocations required to calculate the 40th term of the Fibonacci sequence.

Another problem with recursion is that during a recursive method's execution, if the recursive chain of invocations gets too long, the storage required for the return addresses and CPU register contents can exceed the capacity of the run-time stack. When this happens, the method's execution will produce a runtime `StackOverflowException` error, and unless the exception is caught, the application will be terminated. Below is a summary of the problems with recursion that were discussed in this section:

1. Most programmers need to be trained in how to discover and implement recursive algorithms.
2. The number of recursive invocations issued by a recursive method can make it significantly slower than its interactive counterpart.

3. Methods that implement recursive algorithms can require prohibitively large amounts of RAM storage.
4. Recursive methods are prone to terminating in a `StackOverflowException` error.

One solution to the last three problems will be discussed in Section 9.5.2.

9.5.1 When to Use Recursion

In light of the problems associated with recursion, and considering the fact that recursive algorithms have an iterative counterpart, the question of when we should use recursion in our programs arises. The short answer is when the recursive algorithm significantly reduces the algorithm-discovery and implementation time, and the additional storage requirements and execution time associated with a recursive method are acceptable.

This is not the case for most of the problems presented in Table 9.3. Certainly, the first five problems presented in the table would normally not be coded using recursion because their non-recursive (iterative) solution is easy to discover and simple to code. They were included in the table to facilitate the learning process. On the other end of the spectrum, recursion is usually used to find a greatest common divisor and in the solution to the Towers of Hanoi problem. The following three-line general solution of the Towers of Hanoi is much simpler than a non-recursive solution to the problem:

1. move n-1 rings from tower A to tower C
2. move one ring from tower A to tower B
3. move n-1 rings from tower C to tower B

Other common uses of recursion include the solution of puzzles, such as mazes, the Sudoku puzzle, the Eight Queens problem, the Knight's Tour, the searching and sorting of lists of data, and the drawing of fractals.

Fractals and the Sierpinsky Triangle

Fractals are mathematical or geometric objects that have the property of self-similarity, that is, each part of the object is a smaller or reduced copy of itself. Geometric fractals are implicitly recursive because they begin by drawing a shape and then extend the drawing by repeatedly redrawing the shape at smaller and smaller scales. Generally speaking, the repetitions are infinite, although the drawing process is normally terminated when the shapes become too small to see.

Given an equilateral triangle, the Sierpinsky fractal is produced by creating three equilateral triangles whose vertices are the three original vertices and the midpoints of the sides adjacent to these vertices. This process is then repeated recursively for the three resulting equilateral triangles. The first three steps in this process are shown in Figure 9.15.

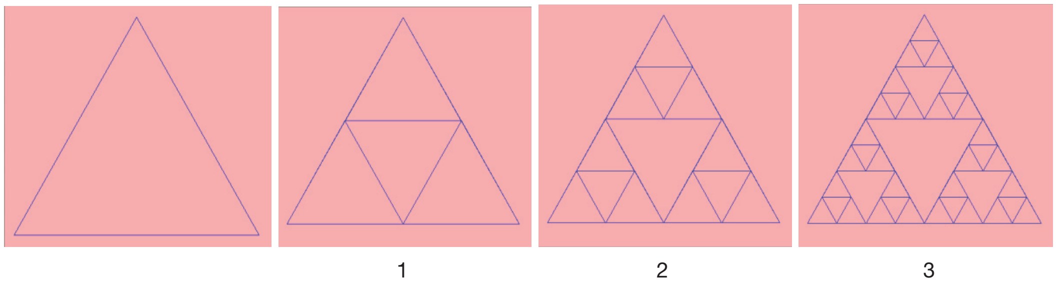


Figure 9.15
The first three steps in the creation of a Sierpinsky Triangle.

The fractal shown in Figures 9.16 is a Sierpinsky Triangle on which the process has been repeated six times. It was created by the application `RecursiveFractal` shown in Figure 9.17. Line 20 of the application invokes the recursive method `drawSierpinsky` (31–47) to draw the fractal. The invocation passes the method the number of times to repeat the Sierpinsky process (plus one for the drawing of the original triangle), the vertices of the original triangle (declared as `Point` objects on lines 8, 9, and 10), and the graphics object `g` because the method draws the triangles (lines 39–41). The vertex at the top of Figure 9.16 is `p1`, the lower left vertex is `p2`, and the lower right vertex is `p3` (lines 8-10).

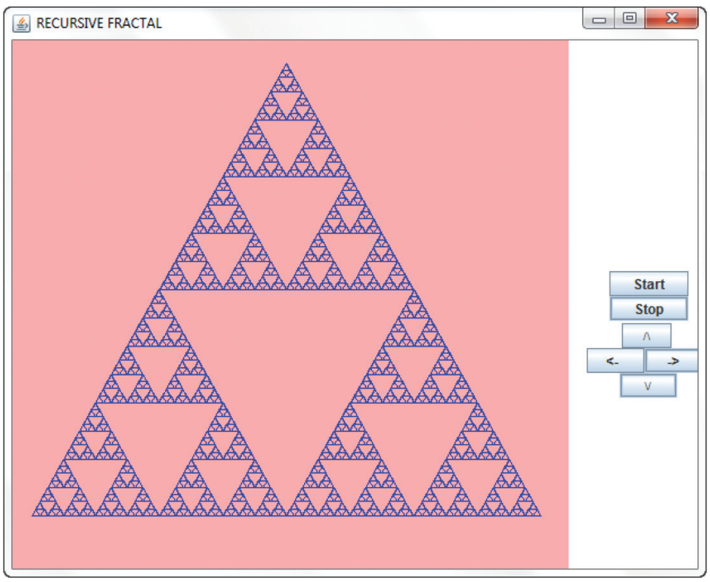


Figure 9.16
A six-iteration Sierpinsky fractal.

Line 34 tests for the base case, which is when the number of invocations of the method has been decremented to zero by line 42. The general solution is in lines 44–46, which recursively invokes the method to draw three triangles. The vertices of these triangles are one of the vertices passed to the method and the midpoints of the lines joining the other two vertices. Each of these three invocations spawns 363 additional invocations of the `drawSierpinsky` method for the fractal drawn by the application. The locations of the midpoints of the sides of the triangle passed as the third and fourth arguments on lines 44–46 are calculated by the `midPoint` method (lines 23–29). The two integer data members of the API `Point` class (x, y) are public data members, which eliminates the need to invoke `set` and `get` methods on lines 26 and 27.

The recursive execution sequence of the lines 44–46 draws all of the blue lines that make up the mostly blue large upper triangle shown in Figure 9.16, before the mostly blue large lower left triangle is drawn, which is drawn before the mostly blue large lower right triangle is drawn.

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3
4    public class RecursiveFractal extends DrawableAdapter
5    {
6      static RecursiveFractal ge = new RecursiveFractal();
7      static GameBoard gb = new GameBoard(ge, "RECURSIVE FRACTAL");
8      static Point p1 = new Point(250, 70); //vertices of the 1st triangle
9      static Point p2 = new Point(25, 460);
10     static Point p3 = new Point(475, 460);
11
12     public static void main(String args[])
13     {
14       showGameBoard(gb);
15     }
16
17     public void draw(Graphics g)
18     {
19       g.setColor(Color.BLUE);
20       drawSierpinsky(8, p1, p2, p3, g);
21     }
22
23     public static Point midPoint(Point p1, Point p2)
24     {
25       Point midPoint = new Point();
26       midPoint.y = p1.y + (p2.y - p1.y)/2;
27       midPoint.x = p1.x + (p2.x - p1.x)/2;
28       return midPoint;
29     }
30
31     public static void drawSierpinsky(int iterations, Point p1, Point p2,
32                                       Point p3, Graphics g)
33     {
34       if(iterations == 0) //base case
35       {
36         return;
37       }
38       //general solution
39       g.drawLine(p1.x, p1.y, p2.x, p2.y); //draw a triangle
40       g.drawLine(p2.x, p2.y, p3.x, p3.y);
```

```
        g.drawLine(p3.x, p3.y, p1.x, p1.y);
        iterations--;
      //reduced problems to draw top, left & right side triangles recursively
        drawSierpinsky(iterations, p1, midPoint(p1,p2), midPoint(p1,p3), g);
        drawSierpinsky(iterations, p2, midPoint(p2,p1), midPoint(p2,p3), g);
        drawSierpinsky(iterations, p3, midPoint(p3,p1), midPoint(p3,p2), g);
    }
  }
```

Figure 9.17
The application **RecursiveFractal**.

9.5.2 Dynamic Programming

Dynamic programming is a technique that can sometimes be used to solve three of the problems associated with recursion: unacceptably long execution times, excessive memory requirements, and a `StackOverflowException` error. It does this by reducing the number of recursive invocations. The basis of this technique is the idea that once a value is computed recursively, it should not be computed again.

For example, Figure 9.18 shows the 25 invocations required to compute the seventh term of the Fibonacci series recursively. During this computation, once the first invocation to compute f3, shown on the lower left side of the figure, completes its execution, the other four invocations to compute f3 shown in the figure could be eliminated because the value of f3 is already known. Once computed, if that value was stored in a static class-level variable, the other four invocations of f3 could be replaced with a new base case that simply returned the value stored in the variable. A similar approach would eliminate three of the four invocations to compute f1, seven of the eight to compute f2, two of the three to compute f4, and one of the invocations to compute f5. The result would be that only seven invocations would be required to compute f7 recursively: one invocation to compute f7, and six more recursive invocations to compute f6, f5, f4, f3, f2, and f1 (see highlights in Figure 9.19).

Figure 9.19 illustrates this improved process of computing f7 = 13 using dynamic programming in the implementation of its recursive algorithm. The invocation of the method to compute f7 still spawns the recursive invocations shown on the far left side of Figure 9.18 to compute f6 though

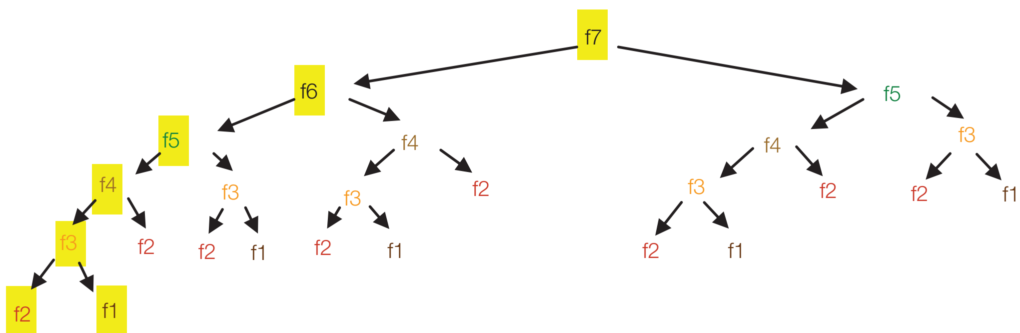


Figure 9.18
The 25 invocations required to compute the seventh term of the Fibonacci sequence recursively.

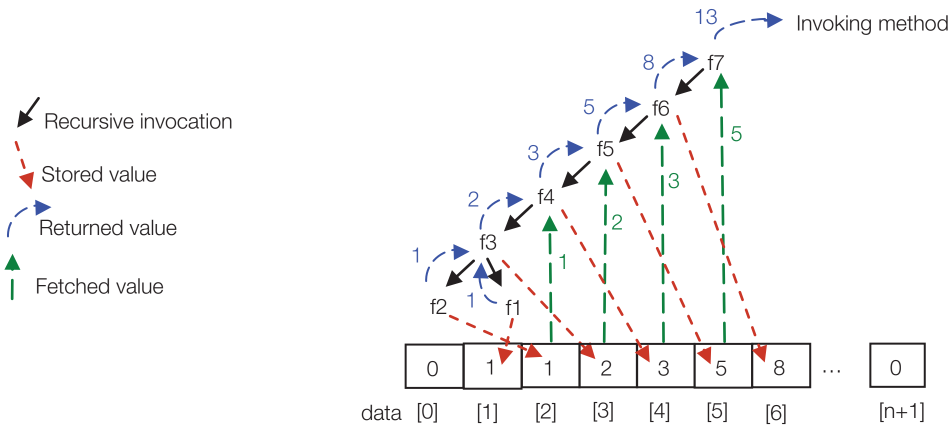


Figure 9.19
The seven invocations required to compute the seventh term of the Fibonacci sequence recursively using dynamic programming.

f1 (black arrows in Figure 9.19). Now, however, after these terms are computed and before they are returned (blue arrows in Figure 9.19), the computed values are stored in an array (red arrows) named `data` in Figure 9.19. In addition, before a recursive invocation is made to compute the value of fi, the i^{th} element of the array is examined to see if it is nonzero. If it is, the value has already been computed, in which case the value is fetched from the array (green arrows in Figure 9.19), and the recursive invocation is not issued.

Referring to the left side of Figure 9.18, this eliminates the need to recalculate f2 when calculating f4, to recalculate f3 when calculating f5, to recalculate f4 when calculating f6, and to recalculate f5 when calculating f7. When dynamic programming is applied to the calculation of the 40^{th} term of the Fibonacci series, the number of invocations of the recursive method is reduced from 204,668,309 invocations to 40 invocations.

The application `FibonacciDynamic` shown in Figure 9.20 contains two implementations of the recursive algorithm to calculate the n^{th} term of the Fibonacci series. One of these implementations incorporates dynamic programming into the algorithm. After the user is asked to enter the number of the term to be calculated, the application invokes both methods and then outputs the calculated values and the number of invocations required to perform the calculations. A typical input and the resulting outputs produced by the application are shown in Figure 9.21.

The output statement that begins on line 18 invokes the non-dynamic implementation, `fib`, and produces the first two lines of output shown in Figure 9.21. It is passed the number of the term to be calculated, which was input and parsed on lines 13–16. The method `fib` (lines 28–42) counts the number of times it is invoked by incrementing the counter variable `invocations` on line 31. The variable is defined as a class-level variable on line 6, so all of the recursive invocations share it.

Line 33 of the method identifies the two defined base cases and returns the base case value, one, on line 35. Lines 38 and 39 invoke the method recursively to compute the two reduced problems, and their resulting sum is returned on line 40. A typical set of recursive invocations generated by this method when it is passed n = 40 and n = 7 are shown in Figures 9.14 and 9.18, respectively.

The output statement that begins on line 23 invokes the dynamic implementation, `fibDynamic`, and produces the last two lines of output shown in Figure 9.21. Before the dynamic version of the method is invoked on line 23, line 21 sets the invocation counter back to zero. Like its non-dynamic counterpart, `fib`, this method (lines 44–75) is passed the number of the term to be calculated (line 44), counts the number of times it is invoked (line 48), and identifies the two defined base cases on line 51. Unlike its dynamic counterpart, line 53 stores the value of these base cases (one) in the array `data` before returning the value. The array `data` is defined as a static class-level array on line 7, and its i^{th} element is used to store the ith term of the sequence after it is calculated.

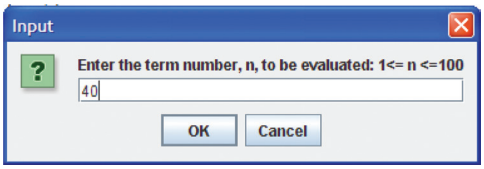
Lines 56–59 check for an additional base case, the value in the n^{th} element of the array `data` is non-zero, in which case it returns the value. This prevents the recursive invocations to calculate fn-1 and fn-2 when the n^{th} term of the sequence has already been calculated and stored in the array.

Line 62 checks the value `data[n-1]` to determine if the value of fn-1, the first reduced problem, has not been calculated. When that is the case, line 64 issues a recursive invocation to calculate it and stores the returned value in element n-1 of the array. Line 64 then sets the variable `rp1` to the calculated value. Lines 67–71 perform the analogous operations for the second reduced problem, fn-2. The general solution, the value of fn, is calculated and returned on lines 72 and 73.

```
1    import javax.swing.*;
2    import java.text.DecimalFormat;
3
4    public class FibonacciDynamic
5    {
6      static long invocations = 0;
7      static long[] data = new long[101];
8
9      public static void main(String[] args)
10     {
11       DecimalFormat f = new DecimalFormat("#,###");
12
13       String s = JOptionPane.showInputDialog("Enter the term number," +
14                                               " n, to be evaluated:" +
15                                               " 1<= n <=100");
16       int n = Integer.parseInt(s);
17
18       System.out.println("fn = " + f.format(fib(n)) +
19                          "\ncalculated making " + f.format(invocations) +
20                          " invocations");
21       invocations = 0;
22       System.out.println();
23       System.out.println("fn = " + f.format(fibDynamic(n)) +
24                          "\ncalculated making " + f.format(invocations) +
25                          " invocations");
26     }
27
28     public static long fib(int n)
29     {
```

```
      long rp1, rp2;
      invocations++;

      if(n == 1 || n == 2) //defined base cases
      {
        return 1;
      }
      else //general solution
      { rp1 = fib(n-1); //calculate first reduced problem
        rp2 = fib(n-2); //calculate second reduced problem
        return rp1 + rp2;
      }
    }

    public static long fibDynamic(int n)
    {
      long rp1 = 0;
      long rp2, gs;
      invocations++;

      //three base cases
      if(n == 1 || n == 2) //defined base cases
      {
        data[n] = 1;
        return 1;
      }
      else if(data[n] != 0) //dynamic programming base case
      {
        return data[n];
      }
      else //general solution
      {
        if(data[n-1] == 0) //calculate f1rst reduced problem
        {
          data[n-1] = fibDynamic(n-1);
        }
        rp1 = data[n-1];
        if(data[n-2] == 0) //calculate second reduced problem
        {
          data[n-2] = fibDynamic(n-2);
        }
        rp2 = data[n-2];
        gs = rp1 + rp2;
        return gs;
      }
    }
  }
```

Figure 9.20
The application **FibonacciDynamic**.



Console output:

fn = 102,334,155
calculated making 204,668,309 invocations

fn = 102,334,155
calculated making 40 invocations

Figure 9.21
Output of **`FibonacciDynamic`**.

9.6 CHAPTER SUMMARY

Recursion is a problem solving tool that can provide a more succinct, elegant solution to a problem than solutions based on other problem solving techniques. Recursive algorithms are implemented within a recursive method, which is a method that repeatedly invokes itself either directly or indirectly until a condition, called a base case, is reached and terminates the sequence of invocations. If the base case is not reached, the method terminates in a stack overflow runtime error because the RAM memory, dedicated to saving the information required to complete each invocation, has been exceeded. While most people don't naturally possess the innate ability to think recursively, they can learn how to do so via a divide-and-conquer methodology reinforced with lots of recursive problem solving practice.

The methodology consists of discovering a problem's base case, reduced problem, and general solution and combining these components into a recursive solution to the problem. The base case is a known, defined, or trivial solution to a similar but usually simpler problem. If the problem involves an integer, n, the base case is usually a version of the problem when n is zero or one. For example, the base case for n^n could be the definition, $x^0 \equiv 1$, or the trivial case $x^1 = x$.

The reduced problem is a simpler version of the problem that satisfies this condition: when the relationship between the problem and reduced problem is repeatedly applied to the reduced problem, it becomes the base case. For example, the relationship between x^n and the candidate reduced problem x^{n-1} is the reduction of the exponent by one. When this is repeatedly applied to the reduced problem x^{n-1}, it becomes x^{n-2}, which becomes x^{n-3}, which eventually becomes both base cases: first x^1 and then x^0. The discovery of the reduced problem is normally the most difficult part of the methodology.

The general solution is the recursive solution to the original problem under the assumption that the solution to the reduced problem is known. For example, if we know how to calculate x^{n-1}, then xn can be calculated by multiplying x^{n-1} by x. Once a valid base case, reduced problem, and general solution have been discovered, the last step of the methodology is to combine them as shown in Figure 9.6. While the methodology cannot be used to formulate all recursive algorithms, by practicing with problems within its domain, we will develop the skills necessary to extrapolate the methodology to the recursive solution of problems beyond its domain.

It is important to understand not only how to use recursion, but when to use it or not use it as well. It should be used when the recursive algorithm significantly reduces the algorithm-discovery and implementation time and when the additional storage requirements and execution time are acceptable. Appropriate problems for recursive solutions are those whose structures are inherently recursive, such as trees, mazes, nested lists, and fractals.

Solutions for some of the common recursive problems are presented in Table 9.4.

Table 9.4
Base Cases, Reduced Problems, and General Solutions for Several Problems With Recursive Solutions

Problem	Base Case	Reduced Problem	General Solution
Factorial of a positive integer, n!	0! ≡ 1	(n – 1)!	n * (n – 1)!
A number x raised to a positive integer, x^n	$x^0 \equiv 1$	$x^{(n-1)}$	$x * x^{(n-1)}$
Sum of the integers from 1 to n	sum ≡ 1 for n = 1	Sum of the integers from 1 to (n – 1)	n + sum of the integers from 1 to (n – 1)
Product of two positive integers, m × n	m * 1 = m	m * (n – 1)	m + m * (n – 1)
Output an n character string s in reverse order	n == 1 output s	Output the last n – 1 characters of string s in reverse order	Output the last n – 1 characters of string s in reverse order. Then output the first character of s
Problems with Multiple Base Cases and Reduced Problems			
Generate the n^{th} term of the Fibonacci Sequence, fn	$f_1 = 1$ and $f_2 = 1$	f_{n-1} and f_{n-2}	$f_{n-1} + f_{n-2}$

Find the greatest common divisor of two positive integers m and n, GCD(m, n), for m > n	if m = n, GCD = m	GCD(m – n, n) and GCD(m, n – m)	if(m > n)
			GCD(m – n, n)
			else
			GCD(m, n – m)

(Contd.)

Problem	Base Case	Reduced Problem	General Solution
A Problem with Multiple Base Cases			
Search a sorted list of items for I	If the list is empty, return not found If the middle item is I, return found	Search a designated sorted sub list for I	if I > middle item, sub list is the items after it, else sub list is items before it. Search the sub list for I
A Problem that Uses the Reduced Problem Twice			
Towers of Hanoi Move n rings from tower A to tower B using tower C	n = 1: Move 1 ring from one tower to another tower	Move n–1 rings from one tower to another tower	Move n–1 rings from A to C Move 1 ring from A to B Move n–1 rings from C to A

Knowledge Exercises

1. True or false:
   a) A method that invokes itself directly or indirectly is said to be recursive.
   b) By definition, zero factorial is equal to one.
   c) Finding the base case is the most difficult part of discovering a recursive algorithm.
   d) Every problem has a recursive solution.
   e) Implementations of recursive solutions can result in a `StackOverFlow` error.
   f) Recursion is useful in drawing fractals.
   g) Implementations of recursive solutions always execute quickly.
2. Which invocation of a recursive method takes the longest time to complete: the first invocation or the last?
3. Which of the flow-chart symbols in Figure 9.6 makes the algorithm it depicts recursive?
4. Calculate the value of 8! using both iterative and recursive techniques.
5. Calculate the value of the sixth term of the Fibonacci sequence using both iterative and recursive techniques.
6. How many terms of the Fibonacci sequence are calculated when the recursive technique (Figure 9.7) is used to calculate the sixth term of the sequence?
7. What is the name of the technique used to reduce the calculations performed in Exercise 6?
8. How many terms of the Fibonacci sequence are calculated when the technique of Exercise 7 is applied to Exercise 6?

9. Give the base case, reduced problem, and general solution used in the recursive solution of the calculation of the sum of the even integers from n up to m, where both n and m are even.

10. Explain why (n-2)! is not a valid reduced problem for n!.

11. Why is it important in the calculation of a factorial that zero factorial be defined as 1?

    (0! ≡1)

12. Use a recursive algorithm for the greatest common divisor to compute the greatest common divisor of 15 and 255.

13. What is the last step in the recursive solution-discovery methodology discussed in this chapter?

14. How many moves would be required in the Towers of Hanoi problem to move six rings? How many for ten rings? How many for n rings?

Programming Exercises

1. Write a recursive method that is passed two integers, x and y, and returns the value of x raised to the power y. Include the method in an application that verifies its functionality.

2. Write a recursive binary search method that is passed an array of integers and an integer value and returns the index of the array that contains the integer passed to it. If the integer is not found, the method returns -1. Include the method in an application that verifies its functionality.

3. Write a recursive method that calculates the sum of the elements of an integer array passed to it. Include the method in an application that verifies its functionality.

4. Write both a recursive and an iterative method to find and return the greatest common divisor of two integer arguments, x and y, passed to it. Include the methods in an application that verifies their functionality.

5. Write a recursive method that returns `true` when the string passed to it is a palindrome. A palindrome is a string that reads the same backwards and forwards. For example, “MADAMIMADAM,” “TOOT,” and “WOW.” Include the method in an application that verifies its functionality.

6. Write a recursive method that outputs the string passed to it in reverse. Include the method in an application that verifies its functionality.

7. Write a graphical application that moves one of the Towers of Hanoi rings from one tower to another every time the up button of the game board is clicked. The program should begin by asking the user to input the number of rings to be relocated, and show the entire solution.

Enrichment

1. Binary trees are recursive structures. Find out how binary tree traversals are performed and explain the recursive algorithms for in-order, pre-order and post-order tree traversals. Why is recursion a suitable approach to tree traversals?
2. How can a recursive algorithm be used to count the nodes in a binary tree?
3. Fractals have the property of self-similarity, and they are often implemented recursively. Research the characteristics of some well-known fractals, such as the Koch snowflake, the Cantor Set, and the Mandelbrot and Julia sets. Explore their algorithms and implementations.
4. Investigate the Heap Sort and Quick Sort recursive algorithms to discover why they are so efficient. Explain briefly how the algorithms work.
5. The Eight Queens and the Knight's Tour problems both have recursive solutions. Investigate these problems and the recursive algorithms used to solve them.
6. Recursion is used in solving mazes. Research a recursive algorithm for traversing a maze and explain the base case, the reduced problem, and the general solution.

References

Barnsley, Michael. *Fractals Everywhere*. San Diego, CA: Academic Press, Inc., 1988.

Mandelbrot, Benoit. *The Fractal Geometry of Nature*. NewYok: W.H. Freeman and Co., 1977.

Peitgen, Heinz-Otto, et al. *Fractals for the Classroom, Strategic Activities*, Part One. New York: Springer-Verlag, 1991.

Peitgen, Heinz-Otto, et al. *Fractals for the Classroom, Complex Systems and Mandelbrot Set*, Vol. 2. New York: Springer-Verlag, 1992.

Roberts, Eric. *Thinking Recursively*. NewYork: John Wiley and Sons, 1986.

Roberts, Eric. *Thinking Recursively with Java,* 20th Anniversary Ed. NewYork: John Wiley and Sons, 2005.

Endnotes

[1] Epp, Suzanna, *Discrete Mathematics with Applications*, 3rd Ed., Brooks Cole, 2004. p, 461.

CHAPTER 10

EXCEPTIONS: A SECOND LOOK

In this chapter

In this chapter, we will discuss the features Java provides within its implementation of the concept of exceptions. This will expand our knowledge of the `try-catch` construct, the API exception classes, and the differences between checked and unchecked exceptions, which were discussed in the context of performing disk I/O. In addition, we will learn how to write methods that throw exceptions when they detect errors, as the methods in the API classes do, and how to create our own exception classes that make our programs more readable. We will also discuss ways of using exceptions to facilitate the implementation of methods that do not normally detect errors and the role of the `finally` clause in exception handling.

After successfully completing this chapter, you should:

- Be able to use the `try-catch` construct to process errors detected by methods you invoke and fully understand the execution path of the construct
- Know when to include a `finally` clause within a `try-catch` construct and its role in the construct's execution path
- Understand how to design and implement algorithms that detect errors and throw exception objects
- Be able to distinguish between the checked and unchecked exceptions as well as to know the differences between these two types of exceptions and their appropriate use
- Be able to define and use new exception classes to develop more readable code
- Know how to use the features of exceptions in non-error detecting methods to facilitate their development
- Understand the translator enforced coding order of multiple `catch` clauses coded within the `try-catch` construct

10.1 AN OVERVIEW

During the development of a program, a significant amount of time is spent testing the methods of the classes that make up the program because one very important programming goal is to produce an error free program. To improve the chances of attaining this goal, the development process begins by dividing a large program into smaller classes, and each class is divided into a collection of small methods because the solution to small problems tends to be less error prone. Then, each method is written and enters an iterative testing process aimed at exposing and eliminating the errors in the method's algorithm and its implementation. (Recall that the programming development process was explained in Chapter 1, and Figure 1.28 illustrates this process.)

Unfortunately, a successful completion of this process does not guarantee that a method will never produce a runtime error or an incorrect result because an unanticipated event could occur during its execution. For example, the user might input a zero divisor into a method that divides two inputs, or the user could direct the method to read data from a file that does not exist. To avoid these failure modes, during the design process we should identify the events that will result in a runtime error or an erroneous result and incorporate code into the method that recognizes them when they occur. For example, before a divisor is used, we should always make sure it is not zero, and we should check for the existence of a file before we attempt to read from the file.

Assuming that this error or failure mode recognition process is properly incorporated into the design of a method's algorithm, we are now faced with the task of deciding what actions to take when these anticipated events occur and incorporating these actions into the method's algorithm. When the method is part of a class that will be imported into someone else's program, this becomes an impossible task because the author of the method does not know what action the user of the class wants to take: terminate the program with an informative error message, give the user an opportunity to correct an erroneous input, or some other course of action.

To solve this dilemma, Java and other programming languages use the concept of exceptions. The name of this concept comes from the analogy of someone asking you to do something you cannot or should not do and you respond, "I take *exception* to your asking me to do that." A method that has determined it is being asked to divide by zero should take exception to that, and a method that has determined that it is being directed to read data from a disk file that does not exist should take exception to that.

The concept of exceptions is based on the idea that a method's algorithm should identify the occurrence of a problem and inform the invoker that it occurred. The burden of what is to be done after the problem occurs is passed on to the invoker. This allows for a recovery from the error that is appropriate to, and implemented by, each application that imports the method's class. For example, one application that received a divide-by-zero exception from a method it invoked could simply inform the program user that division by zero is not possible and terminate, while another application invoking the same method can give the user an opportunity to reenter the divisor.

Exception Terminology: A Baseball Analogy

As mentioned in Chapter 4, the terminology of exceptions comes from a communication analogy between two persons that involves a baseball. One person wraps a message around the ball and then throws the ball to the other person who catches it, unwraps the message, and reads it. Applying the analogy to the communication of error messages, when a method's algorithm detects a problem, it wraps an error message around an exception object and *throws* the object to the code block that invoked it. The code block can *catch* the thrown exception object, unwrap the error message, and process it.

To extend the analogy a bit further, if a person chooses not to catch a thrown ball, some other ordered collection of people can elect to catch it. The runtime environment maintains a history, or the order of a method's invocation sequence. For example, if the method `main` invoked method `A`, and method `A` invoked method `B`, and method `B` invoked method `C`, then `C`'s invocation sequence would be `main`, `A`, `B`. Invocation sequences are stored in a structure called the runtime stack. If method `C` invoked method `D` during its execution, then `C` would be added to, or pushed onto, the runtime stack before `D`'s execution began. The runtime stack would then contain the invocation sequence `main, A, B, C`.

To return to our analogy, if `C` chooses not to catch an exception that is thrown by the method it invoked during its execution, `D`, then each of the methods on the runtime stack are given a chance to catch it in the reverse order in which they were invoked (last in, first out). In our original example, `B` would be given a first chance to catch the thrown exception object, then `A`, and then `main`. We say that the exception object propagates up the runtime stack. If none of the methods on the runtime stack catch the exception, the Java Runtime environment catches and processes it.

NOTE *All uncaught exception objects are caught by the Java Runtime environment.*

After catching an exception, the Java Runtime environment outputs an error to the System console that includes the unwrapped message and then terminates the program. This commonly occurs during the development process when a coding error results in a method being invoked that uses a reference variable that contains a `null` value or an array index is generated that is beyond the bounds of the array. The console outputs, produced by the Java Runtime environment when these two errors occur, are familiar to most of us:

- exception in thread "main" java.lang.NullPointerException
- exception in thread "main" Java.lang.ArrayIndexOutOfBoundsException: -23

Using the techniques discussed in Chapter 4, which are expanded in this chapter, we can catch thrown exceptions and then either continue the program's execution after exceptions are caught or bring the program to a more informative "soft-landing" termination.

10.2 JAVA'S EXCEPTION CLASSES AND EXCEPTION OBJECTS

Figure 10.1 shows some of the API classes that are direct or indirect subclasses of the class `Throwable`. Instances of all of these classes are considered Java exception objects that can be thrown and caught by `try-catch` blocks. Error messages can be wrapped around these objects before they are thrown and unwrapped and processed when they are caught.

NOTE *In Java, an exception object is an instance of the API class* `Throwable` *or an instance of a class that is a direct or indirect subclass of it.*

Checked and Unchecked Exceptions Classes

As shown at the top of Figure 10.1, the classes `Exception` and `Error` are the two direct subclasses of `Throwable`. The class `Error` and all of its descendants are *unchecked* exception classes. Conversely, the class `Throwable` and the class `Exception` are *checked* exception classes. All direct and indirect subclasses of the class `Exception` are also checked exception classes except for the class `RunTimeException` and its descendants.

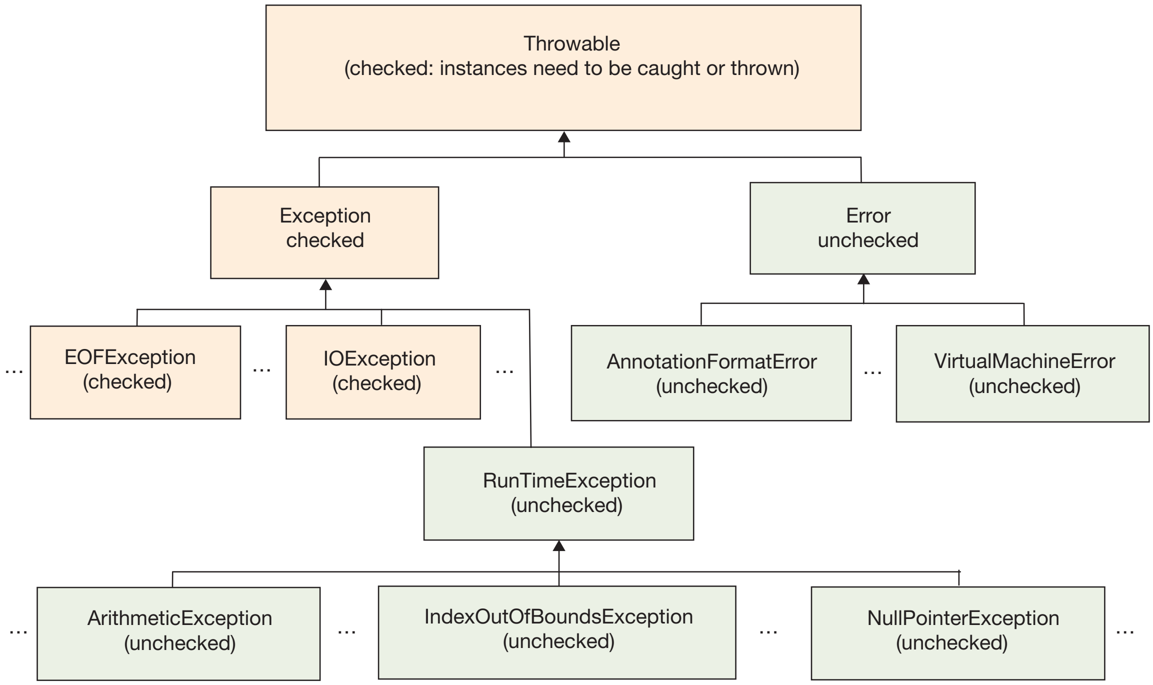


Figure 10.1
A subset of the exception classes included in the Java API and their inheritance chains.

When we invoke a method whose algorithm uses an instance of a *checked* exception to communicate to its invoker that an error has occurred, the invoking method must catch the object or include a `throws` clause in its heading to inform the translator that it is intentionally going to ignore the error detected by the method it invoked. The syntax of a `throws` clause is the keyword throws followed by one or more exception class names separated by commas. The exception object classes coded in the clause must be the type of the ignored checked exception or a direct or indirect super class of it.

As shown in Figure 10.1, the class `IOException` is a checked exception class. Instances of this class and its descendants are thrown by the methods discussed in Chapter 4 that perform disk I/O. That is why the two disk I/O programs shown in Figures 4.20 and 4.23 had to include a `throws` clause at the end of the main method's signature (line 6 of Figure 4.20) or perform the disk I/O from within a `try-catch` construct (lines 39–66 of Figure 4.23) that could catch the thrown checked exception object. If the disk I/O methods threw instances of *unchecked* exceptions, not including either a `throws` clause or the `try-catch` construct in the method that invoked them would be syntactically correct.

NOTE *Thrown checked exceptions must be caught, or the signature of the invoking method must include a* `throws` *clause.*

Errors that cause instances of the class `Error` or its descendants to be thrown are considered abnormal conditions; they should normally not occur. They are not caused by programming errors, an I/O error, or something that can be dealt with within an application. When they do occur, they are best processed by the Java Runtime environment, the catcher of all uncaught exceptions. Runtime errors that generate checked exceptions are considered to be situations that can be dealt with by a method within an application and are serious enough that the translator requires that they either are dealt with or that the method indicates that it is intentionally ignoring the error via a `throws` clause.

Errors that cause instances of the class `RunTimeError` or its descendants to be thrown are considered errors that will be eliminated during the testing phase of the application's development, and therefore, the translator does not require that the application process these errors or indicate that they are intentionally being ignored via a `throws` clause. In situations where the programmer feels that erroneous input or other non-programming error-related events could cause these unchecked exception objects to be thrown, a `try-catch` construct should be included in the portions of the application where these events could occur.

10.3 PROCESSING THROWN EXCEPTIONS

Exceptions are processed using a `try-catch` construct. The construct consists of a `try` clause that is immediately followed by a `catch` clause. The statements associated with each of these clauses are always enclosed in a set of brackets, even if there is only one statement associated with them. For this reason, they are commonly referred to as `try` and `catch` blocks.

The `try` block is used to detect thrown exceptions, and the `catch` block is used to process the errors that produced the exceptions. As illustrated in Figure 10.2, one `try` block can be followed by multiple `catch` blocks, and the `catch` blocks must immediately follow the `try` block. Coding statements in between any of the blocks is a syntax error. In the absence of a `finally` block, which will be discussed later in this chapter, at least one `catch` block must be included in the construct.

```
try
{
  //try to execute the statements in this statement block
}
catch(ExceptionClass1 thrownObject1)
{
  //statements to execute when an ExceptionClass1 object is thrown
}
```

```
catch(ExceptionClass2 thrownObject2)
{
  //statements to execute when an ExceptionClass2 object is thrown
}
          :
          :
catch(ExceptionClassN thrownObjectN)
{
  //statements to execute when an ExceptionClassN object is thrown
}
```

Figure 10.2
Syntax of the **`try-catch`** construct.

The first line of each `catch` block includes a single parameter that is a reference to an exception object. Its type must be `Throwable` or a descendent of that class. When multiple `catch` blocks are included in the construct, each block must contain a different parameter type.

The execution path of the `try-catch` construct is shown in Figure 10.3. The statements in the `try` block are executed until one of them causes an exception, at which point the execution of the

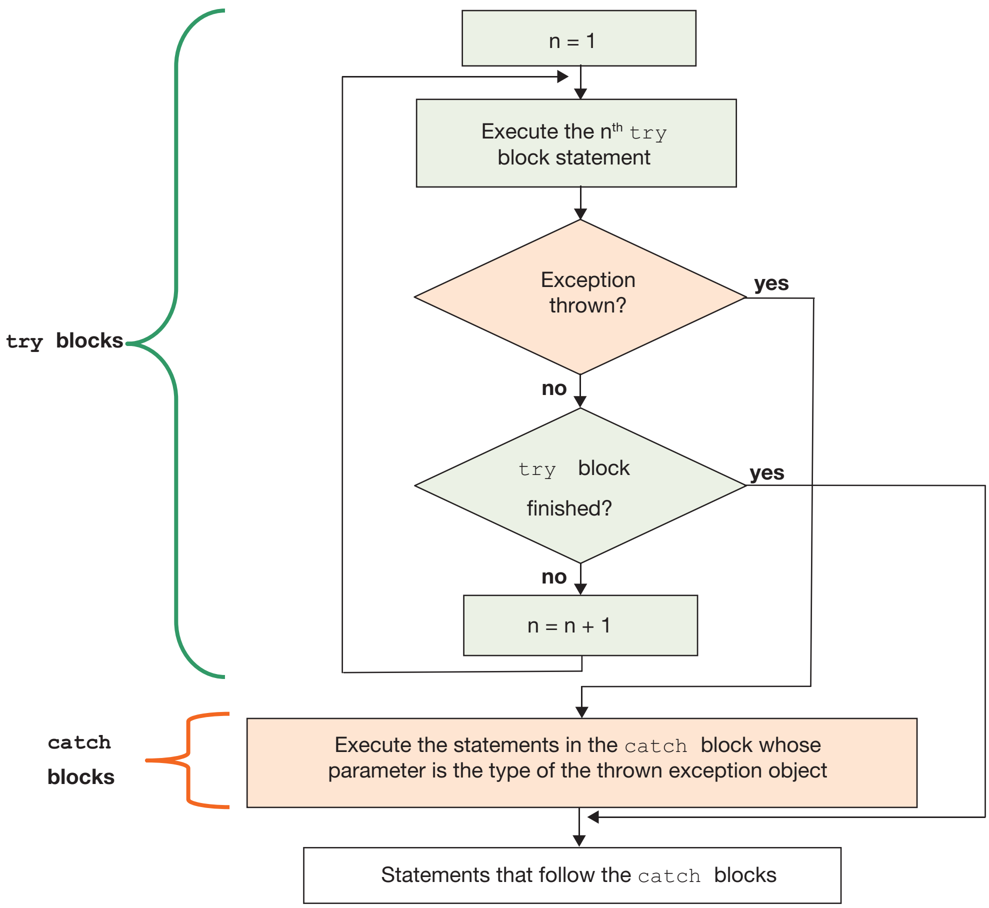


Figure 10.3
The execution path of the **`try-catch`** construct.

`try` block statement terminates, and the statements in the `catch` block whose parameter matches the type of the thrown exception object begins execution.

NOTE *A thrown exception object can be caught by a* `catch` *block whose parameter type is a direct or indirect super class of the thrown exception type.*

After the statements in the `catch` block complete their execution, the statements that follow the `catch` blocks begin execution. If the type of the thrown exception object does not match any of the parameter types in the `catch` blocks, and it is an unchecked exception, the statements that follow the `catch` blocks are executed. If an uncaught exception is a checked exception, the method terminates.

Figure 10.4 presents the application `ProcessingExceptions` that calculates the quotient and remainder of two input numbers and gives the user three opportunities to correct the erroneous input of a zero divisor. The `for` loop that begins on line 18 provides the three attempts to perform the division. Its statement block (line 19–42) includes the calculations, the error detection of a zero divisor, and the error processing.

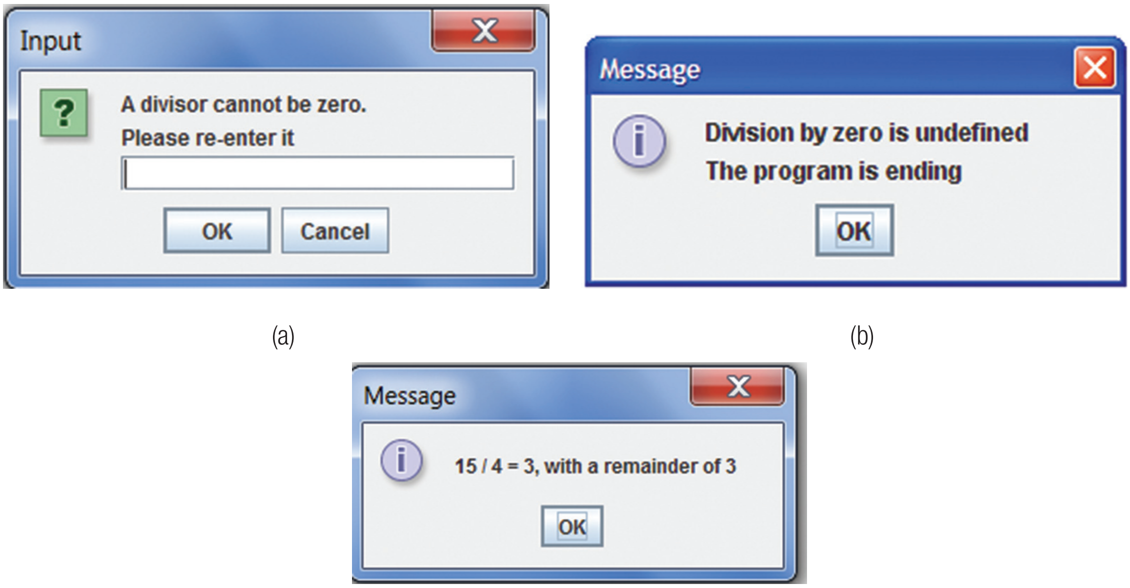
Line 22 performs the division of the two numbers initially input and parsed on lines 12–16. Java's divide operator, used on line 22, throws an `ArithmeticException` object whenever the operation's divisor is zero. Because the statement is inside the `try` block that begins on line 20, whose `catch` block processes this type of exception (line 26), when the error occurs, the `if-else` statement that begins on line 28 executes. If three attempts to perform the division have not been made, the `if` statement's code block accepts another value of the divisor (Figure 10.5a), and the next iteration of the loop begins. After the third failed attempt to perform the division, the `else` clause's code block executes (lines 35–40) outputting an error message (Figure 10.5b), and line 39 terminates the program.

When the division performed on line 22 is successful, the `try` block completes its execution, and lines 43–45 output the result of the division (Figure 10.5c). The `break` statement at the end of the `try` block (line 24) should not be interpreted as breaking out of the `try-catch` construct; rather, its action is to terminate the `for` loop. The process of exiting a `try-catch` construct, illustrated in Figure 10.3, does not involve a `break` statement; this statement is used to terminate a switch or loop construct. If the `try-catch` construct were not coded inside the `for` loop that begins on line 18, the `break` statement on line 24 would have produced a translation error.

```
1    import javax.swing.*;
2
3    public class ProcessingExceptions
4    {
5      public static void main(String[] args)
6      {
7        String s;
8        int a, b;
9        int quotient = 0;
10       int remainder = 0;
```

```

    s = JOptionPane.showInputDialog("This program calculates a / b " +
                                    "\nEnter the value of a" );
    a = Integer.parseInt(s);
    s = JOptionPane.showInputDialog("Enter the value of b" );
    b = Integer.parseInt(s);

    for(int i=1; i<=3; i++) //three attempts to divide a and b
    {
      try
      {
        quotient = a / b; //throws an ArithmeticException
        remainder = a % b;
        break; //ends the for loop and Line 43 executes next
      }
      catch(ArithmeticException e)
      {
        if(i != 3) //three attempts to divide have not been made
        {
          s = JOptionPane.showInputDialog("A divisor cannot be zero." +
                                          "\nPlease re-enter it");
          b = Integer.parseInt(s);
        }
        else
        {
          JOptionPane.showMessageDialog(null, "Division by zero " +
                                              "is undefined \n" +
                                              "The program is ending");
          System.exit(0); //terminate the program
        }
      } //end of the try-catch construct
    }//end of for loop
    JOptionPane.showMessageDialog(null, a + " / " + b + " = " +
                                        quotient + ", with a " +
                                        "remainder of " + remainder);
  }
}
```

Figure 10.4
The application **ProcessingExceptions**.



(a) (b) (c)

Figure 10.5
Outputs produced by the application **`ProcessingExceptions`**.

While it is true that the error checking performed with the `try-catch` construct used in the application shown in Figure 10.4 could be performed using an `if-else` statement, in most cases, the use of exceptions make our code more readable and simpler to code when more than one error can occur within in a code block.

The application `MultipleCatchBlocks` presented in Figure 10.6 is a modified version of the application presented in Figure 10.4. This version of the program uses a second `catch` block to verify that the two user inputs are integers, and, in the interest of brevity, ends after it processes this or the divide-by-zero error. Determining if the input string is an integer is a relatively simple thing to do because the `Integer` class's `parseInt` method throws a `NumberFormatException` object when the string passed to it contains any character other than a digit or a leading plus or minus sign. (The API online documentation of every method included in the API identifies the exceptions each method's throws.)

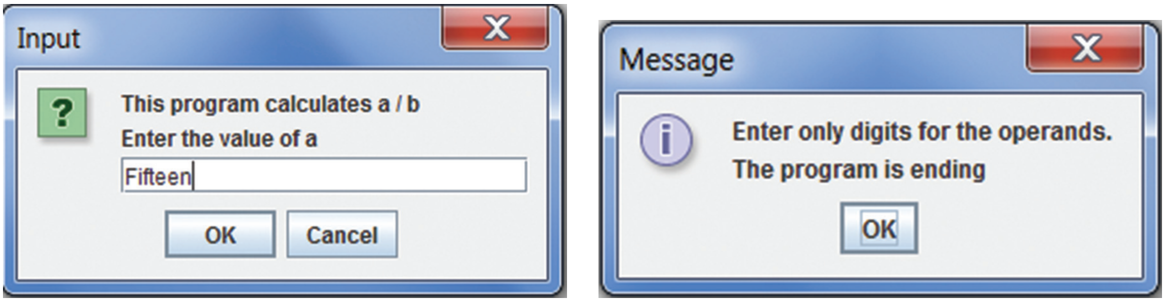
To take advantage of this fact, the parsing of the inputs has been moved into the program's `try` block (lines 17 and 19 in Figure 10.6), and the program's second `catch` block (lines 30–36) processes a `NumberFormatException` object. As shown in Figure 10.2, there is no limit to the number of `catch` blocks that can be associated with one `try` block.

Figure 10.7a shows an erroneous (non-integer) user input, which terminates the `try` block on line 7. Figure 10.7b shows the resulting error-message output by lines 32–34 before the program is terminated by line 35.

```java
import javax.swing.*;

public class MultipleCatchBlocks
{
  public static void main(String[] args)
  {
    String sa, sb;
    int a = 0;
    int b = 0;
    int quotient = 0;
    int remainder = 0;

    try
    {
      sa = JOptionPane.showInputDialog("This program calculates " +
                                       "a / b\nEnter the value of a");
      a = Integer.parseInt(sa); //throws a NumberFormatException
      sb = JOptionPane.showInputDialog("Enter the value of b" );
      b = Integer.parseInt(sb); //throws a NumberFormatException
      quotient = a / b;
      remainder = a % b;
    } //end of the try block
    catch(ArithmeticException e) //process divide by zero
    {
      JOptionPane.showMessageDialog(null, "Division by zero " +
                                          "is undefined. \n" +
                                          "\nThe program is ending");
      System.exit(0);
    } //end of the first catch block
    catch(NumberFormatException e) //process non-integer input
    {
      JOptionPane.showMessageDialog(null, "Enter only digits " +
                                   "for the operands." +
                                   "\nThe program is ending");
      System.exit(0);
    } //end of the second catch block
    JOptionPane.showMessageDialog(null, a + " / " + b + " = " +
                                        quotient + ", with a " +
                                        "remainder of " + remainder);
  }
}
```

Figure 10.6

The application **MultipleCatchBlocks**.



(a) (b)

Figure 10.7
Outputs produced by the application **`MultipleCatchBlocks`**.

Unwrapping Error Messages

The `Throwable` class's `getMessage` method can be used to unwrap an error message. The method returns a string containing the message that was "wrapped around" the exception object that invoked it. The following code fragment outputs the message contained in the object caught by the `catch` statement:

```
catch (RuntimeException e)
{
  System.out.println(e.getMessage());
}
```

10.3.1 Non-error Checking Use of Exceptions

There are times when it is advantageous to use a `try-catch` construct not to detect and process errors but to efficiently identify and process data. Consider the case when an input string contains a mix of characters and numerics, and we want to process just the numerics. For example, add up the numbers in the string:

Please add up 3.4 plus 5 plus -2 OK?

After isolating the tokens (entities separated by white space) in the input string, we can avoid processing (totaling) the non-numeric tokens by using the `NumberFormatException` thrown by the `Double` class's `parseDouble` method to bypass the totaling algorithm. We are effectively using exceptions to identify the numeric tokens in the string. Normally, the `catch` block in this type of application is empty and is only included in the application to make it syntactically correct.

The application `ParsingNumerics`, shown in Figure 10.8, accepts an input string and outputs the sum of the numerics contained in the string. A typical input and output is shown in Figure 10.9.

Line 14 of the application uses the `String` class's `split` method to place the tokens of the string, input on line 12, into the elements of the `String` array `tokens`. The right side of the totaling algorithm on line 19 uses the loop variable of the `for` loop (that begins on line 15) to attempt to parse each token into a `double`. Because line 19 is inside a `try` block (lines 17–21), when the token is nonnumeric, the `parseDouble` method invoked on that line throws a `NumberFormatException` object, and line 19 does not complete its execution. The exception is caught by the `catch` statement (line 22), and the next iteration of the `for` loop begins.

When the parsed token is a numeric, line 19 completes its execution by adding the parsed token to the current total. Then, the token and a plus sign are concatenated into the string `numerics`. When the loop ends, line 27 replaces the rightmost plus sign in the string `numerics` with an equal sign. Then, the string and the total of the numeric values are output to a message box on line 28.

```
1    import javax.swing.*;
2
3    public class ParsingNumerics
4    {
5      public static void main(String[] args)
6      {
7        String input;
8        String[] tokens;
9        double sum = 0;
10       String numerics = "";
11
12       input = JOptionPane.showInputDialog("Enter a String containing " +
13                                           "numerics");
14       tokens = input.split(" +");
15       for(int i = 0; i<tokens.length; i++)
16       {
17         try
18         {
19           sum = sum + Double.parseDouble(tokens[i]); //only numeric added
20           numerics = numerics + tokens[i] + " + "; //build output string
21         }
22         catch(NumberFormatException e) //non-numeric
23         {
24           //prevents termination of application when exception is thrown
25         }
26       } //replace the last plus sign with an equals and produce the output
27      numerics = numerics.substring(0, numerics.length() - 2) + "= ";
28      JOptionPane.showMessageDialog(null, numerics + sum);
29    }
30 }
```

Figure 10.8
The application **`ParsingNumerics`**.

(a) (b)

Figure 10.9
An input to the application **`ParsingNumerics`** and the resulting output.

10.3.2 The finally Clause

A `try-catch` construct can include a `finally` clause. When it does, as shown in Figure 10.10, the `finally` clause must immediately follow the last `catch` clause. Statements cannot be coded between the last `catch` clause and the `finally` clause. It is similar to the other two clauses in that it must contain a code block that could be empty or contain one or more statements.

```
try
{
   //try to execute the statements in this statement block
}
catch(ExceptionClass1 thrownObject1)
{
  //statements to execute when an ExceptionClass1 object is thrown
}
catch(ExceptionClass2 thrownObject2)
{
  //statements to execute when an ExceptionClass2 object is thrown
}
          :
          :
catch(ExceptionClassN thrownObjectN)
{
  //statements to execute when an ExceptionClassN object is thrown
}
finally
{
  //code executed after one of the above blocks completes execution
}
```

Figure 10.10
Syntax of the **`try-catch-finally`** construct.

The `finally` block's function is to implement the tasks associated with the construct that should be performed after the `try` block or one of the `catch` blocks completes its execution. The finally block *always* executes. Figure 10.11 shows the execution path of the try-catch-finally construct.

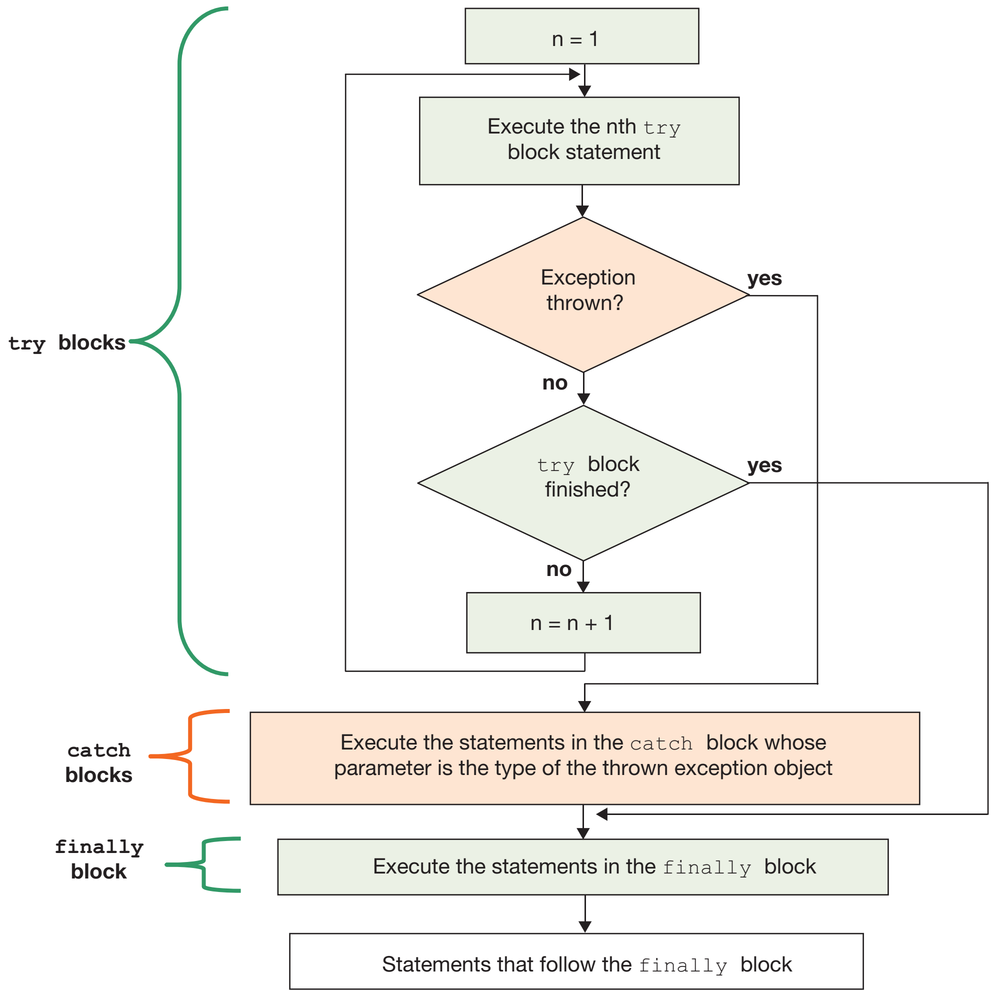


Figure 10.11
The execution path of the **try-catch-finally** construct.

The most common use of the finally clause is to close out the processing of the construct. Because the clause always executes, this processing includes the tasks that are common to, and would otherwise be implemented at, the end of the try block and all of the catch blocks. A proper understanding of the use of the finally block is that if it is not included in the construct, its code would have to be pasted into the end of the try block and into the end of all of the catch blocks. A good example of its use is to close all of the files that were opened during the execution of the other portions of the construct.

Java supports a try-finally construct. It is an error-detection and processing construct that does not contain a catch clause. The finally clause must immediately follow the try clause; statements cannot be coded in between them. After the try block completes its execution, the finally block executes, whether or not an exception is generated during the execution of the try block. When the finally block completes its execution, if an exception was not generated during the execution of the try block, the statements in the method that follow the finally block execute. If an exception was generated during the execution of the try block, the method of which it is a part terminates *after* the finally block completes execution. The statements that follow the finally block do not execute because the thrown exception was not caught. If the exception is a checked exception, the method's signature must contain a throws clause, or it will not translate.

The try-finally construct is used when the thrown exception is to be propagated up the runtime stack. The finally clause's code block can be empty, or it can contain the residual processing to be performed before the construct completes execution. For example, files being written to in the try block are closed or an output is performed. The exception thrown during the execution of the try block is not propagated up the runtime stack until the finally block completes its execution.

The application TryFinally, presented in Figure 10.12, contains a method named appendDataItem (lines 20–35), which is invoked on line 11, to write the string data to a disk file. The method uses a try-finally construct (lines 24–33) to make sure that the disk file to which it writes is closed when it ends execution. The signature of the method (lines 20 and 21) contains two string parameters; the first is the name of the file, and the second is the data item to be written to the file. The method does not contain a catch clause because it defers the decision as to what to do when a file output error occurs to its invoker. As a result, its signature contains a throws clause.

Line 28 writes to the file, and line 32 closes the file. Because the file is closed inside the finally block, it is executed whether or not an error occurs. Line 34 informs the user that the disk write was successful. It only executes if an exception is not thrown in the try block because the method does not include a catch clause. The console output shown at the bottom of Figure 10.12 was produced by line 34 after a successful disk write.

If the disk write is unsuccessful, the method ends after the finally block completes its execution. The thrown exception propagates up the runtime stack to the method main, where it is caught by line 13; then line 15 perform its output, and the program is terminated (line 16).

The declaration of the variable fileOut, which is assigned the address of a PrintWriter object on line 27, has to be coded outside of the try block (line 23). Otherwise, it would not be visible to and could not be used by line 32 to close the file.

```
1    import java.io.*;
2    import javax.swing.*;
3
4    public class TryFinally
5    {
6      public static void main(String[] args)
7      {
8        String data = JOptionPane.showInputDialog("Enter a data item");
9        try
10     {
```

```
      appendDataItem("dataFile.txt", data);
    }
    catch(IOException e)
    {
      System.out.println("There were problems writing to the file");
      System.exit(0);
    }
  }

  public static void appendDataItem(String fileName,
                                    String dataItem) throws IOException
  {
    PrintWriter fileOut = null;
    try
    {
      FileWriter fileWriterObj = new FileWriter(fileName, true);
      fileOut = new PrintWriter(fileWriterObj);
      fileOut.println(dataItem);
    }
    finally
    {
      fileOut.close();
    }
    System.out.println("The data was written to the file");
  }
}
```

Program output:

The data was written to the file

Figure 10.12
The application **`TryFinally`** and the output it produces.

10.4 THE THROW STATEMENT AND ERROR MESSAGES

When designing an algorithm for a method, we should always consider what could go wrong and include a strategy in the algorithm to detect the error. Even the design of a game piece's `set` method could include the ability to detect when the value passed to it is outside a valid range. For example, the `setX` method in a game piece's class could include an `if` statement to make sure the game piece's new x coordinate is within the boundaries of the game board: `minX` ≤ `newX` ≤ `maxX`, as shown below.

```
public void setX(int newX)
{
  if(newX < minx || newX > maxX)
  {
    // Take some action
  }
```

```
    else
    (
      x = newX;
    }
  }
```

To complete the algorithm, we must decide what action to take when the error is detected; that becomes the code of the `if` statement's code block. One strategy may be to set the x coordinate to a value that is within the bounds of the game board. However, this strategy may not be acceptable to all applications that use, or will use, this type of game piece. Even if one strategy can be found, such as asking the game player to re-enter the location of the game piece, one application may require a mouse click on the game board to identify the correct location and another may ask that the new location be entered via a dialog box.

When it is the case that one strategy may not suit the requirements for all applications that use instances of the game piece, the best strategy is for the method to throw an exception object whose message provides as much information as possible about the cause of the error. Then, each application can catch the object, examine the error message, and implement a recovery strategy that best suits the application.

The throw Statement

A method uses a `throw` statement to throw an exception. The statement begins with the keyword throw, which is followed by a reference to an exception object:

```
throw exceptionObject;
```

An exception object is an instance of an exception class, which is the API class `Throwable` or any of its direct or indirect subclasses. Referring to Figure 10.1, if we want the thrown exception to be an unchecked exception, the object thrown should be an instance of the class `RunTimeException` or one of its descendants. Checked exception objects are instances of the class `Exception` or one of its descendants other than `RunTimeException`.

NOTE *When a method throws a checked exception object, its signature must include a* `throws` *clause containing the name of the object's class or one of its ancestor classes.*

Once a decision has been made as to whether the exception will be checked or not, it is good programming practice to declare the exception object to be an instance of an exception class whose name best describes the error that was detected. This makes our programs more readable. For example, if a null reference was detected, the `NullPointerException` class would normally be chosen.

NOTE *A* `throws` *clause begins with the keyword* **`throws`** *followed by the name of one or more exception classes separated by commas.*

Creating Error Messages

An error message is created by passing a string containing the message to an exception class's one-parameter constructor when an exception object is created. The error is wrapped around, or contained in, the object when it is created. The `setX` method, shown in Figure 10.13, uses this approach on lines 5 and 9 to communicate which game board edge of a 500-pixel-wide board would have been breached by a 40-pixel-wide game piece. Exception object messages can be accessed using the `Throwable` class's `getMessage` method.

```
1   public void setX(int newX)
2   {
3     if(newX > 460) //beyond game board's right edge
4     {
5       throw new RuntimeException("Beyond the board's RIGHT edge");
6     }
7     if(newX < 6) //beyond game board's left edge
8     {
9       throw new RuntimeException("Beyond the board's LEFT edge");
10    }
11    x = newX;
12  }
```

Figure 10.13
A game piece's **setX** method that throws an exception containing a descriptive message.

The signature of the method shown in Figure 10.13 does not include a `throws` clause because a `RuntimeException` is an unchecked exception class. An alternative to the nameless objects created on lines 5 and 9 would be to use named exception objects, but the nameless-object approach used in the figure tends to be more readable.

```
// named exception object alternative to Line 5 of Figure 10.13
RuntimeException e =  new RuntimeException("Beyond the board's RIGHT");
throw e;
```

NOTE *When a string is not passed to an API Exception class's constructor a default error message is used.*

Execution Path of the `throw` Statement

After a `throw` statement is executed within a method, the execution of the method ends, and the exception object propagates up the invocation sequence stored in the runtime stack. If the method is a non-void method, a value is not returned from the method.

On the client side, if the method was invoked inside a `try` block, the `try` block's execution ends, and the execution sequence of the `catch` and `finally` blocks begins. If the exception is not caught by the invoking method, it continues up the invocation sequence stored in the runtime stack until it is caught. If the Java Runtime environment catches the exception object, the error message is displayed to the system console, and the application is terminated.

The class `BoxedSnowman2` is shown in Figure 10.14. Its `set` methods that begin on lines 37 and 53 throw a `RunTimeException` object when the value passed to them locates a portion of a snowman beyond the boundaries of the game board. The `if` statements on lines 39 and 43 of the `setX` method detect that an erroneous value of the snowman's `x` coordinate was passed into the method's parameter `newX`. When this is the case, lines 41 and 45 throw a nameless `RunTimeException` object containing appropriate error messages.

The throwing of the exception objects terminates the execution of the `setX` method, leaving the `x` data member unchanged. If an erroneous value is not detected, line 47 performs the normal function of a `setX` method: setting the value passed to the method into the object's `x` data member. Similar modifications have been made to the standard coding of the class's `setY` method (lines 53–64).

The class's three-parameter constructor (lines 9–19) has also been modified to only create snowmen that are completely on the game board. In the interest of brevity, this is accomplished by invoking the class's `setX` and `setY` methods (lines 12 and 13) to store the object's location in its x and y data members. If either of these method invocations, coded inside the `try` block that begins on line 11, produces a thrown exception (because the `intialX` or `intialY` value passed to the constructor is invalid), the constructor does not complete its execution, and it does not return the address of a newly created snowman. Because the constructor does *not* contain a `catch` clause, a thrown exception propagates its way up the invocation sequence stored in the runtime stack. The empty `finally` clause, lines 15-17, is included to make the use of the `try` clause syntactically correct.

```
1    import java.awt.*;
2
3    public class BoxedSnowman2
4    {
5      private int x = 8;
6      private int y = 30;
7      private Color hatColor = Color.BLACK;
8
9      public BoxedSnowman2(int intialX, int intialY, Color hatColor)
10     {
11       try
12       { setX(intialX); //x = intialX;
13         setY(intialY); //y = intialY;
14       }
15       finally
16       {
17       }
18       this.hatColor = hatColor;
19     }
20     public void show(Graphics g) //g is the game board object
21     {
22       g.setColor(hatColor);
```

```
      g.fillRect(x + 15, y, 10, 15); //hat
      g.fillRect(x + 10, y + 15, 20, 2); //brim
      g.setColor(Color.WHITE);
      g.fillOval(x + 10, y + 17, 20, 20); //head
      g.fillOval(x, y + 37, 40, 40); //body
      g.setColor(Color.RED);
      g.fillOval(x + 19, y + 53, 4, 4); //button
      g.setColor(Color.BLACK);
      g.drawRect(x, y, 40, 77); //inscribing rectangle
    }
    public int getX()
    {
      return x;
    }
    public void setX(int newX)
    {
      if(newX > 460)
      {
        throw new RuntimeException("x is beyond the board's RIGHT");
      }
      if(newX < 6)
      {
        throw new RuntimeException("x is beyond the board's LEFT");
      }
      x = newX;
    }
    public int getY()
    {
      return y;
    }
    public void setY(int newY)
    {
      if(newY < 30)
      {
        throw new RuntimeException("y is beyond the board's TOP");
      }
      if(newY > 423)
      {
        throw new RuntimeException("y is beyond the board's BOTTOM");
      }
      y = newY;
    }
  }
```

Figure 10.14
The class **`BoxedSnowman2`**.

The application `ThrowingExceptions`, shown in Figure 10.15, displays a `BoxedSnowman2` object on the game board at a user specified location, which can be moved around the game board using the keyboard cursor control keys. The application uses `try-catch` constructs on lines 24–32 and lines 50–82, to ensure that both the initial and subsequent game board locations of the snowman are within the boundaries of the board.

The loop that begins on line 16 and ends on line 33 asks the user for the initial location of the snowman and creates a snowman at that location. It continues to execute until the user enters a valid initial (x, y) `BoxedSnowman` location: one that would position the entire snowman on the game board. The user-specified x and y coordinates, input and parsed on lines 17–22, are passed to the `BoxedSnowman2` class's three-parameter constructor on line 26. Because line 26 is inside of a `try` block (that begins on line 24), if the `setX` or `setY` methods in the `BoxedSnowman2` class invoked by its constructor throws an exception, line 26 will not complete its execution. As a result, the snowman is not created.

The thrown exception is caught on line 29; line 31 outputs the exception message, and the next loop iteration begins. An erroneous user input and the corresponding output are shown at the top of Figure 10.16. When the user enters a valid location for the snowman, line 26 places the address of the newly created snowman in the variable s1, and the truth value of the Boolean variable `correctXY` used on line 16 is set to `true` (line 27), which terminates the loop. Line 35 causes the `draw` call back method to execute, and line 43 displays the snowman on the game board (Figure 10.16c).

Lines 46–83 implement the game environment's `keyStruck` call back method, which is used to move the snowman around the game board. Inside of its `switch` statement that begins on line 52, lines 57, 63, 69, and 75 invoke the `BoxedSnowman2` class's `set` methods to move the snowman when one of the cursor control keys is struck. If these methods determine that the new x or y coordinate passed to them is invalid, they do not change the snowman's location and terminate after throwing a `RuntimeException` object.

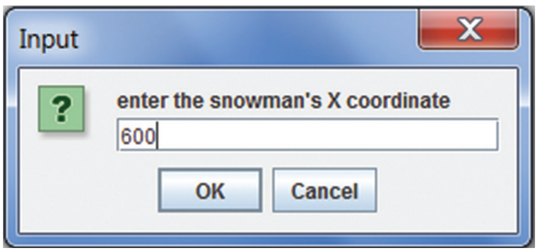
Because the `set` method invocations are coded inside a `try-catch` construct (lines 50–82), the thrown exception is caught on line 79. Its message is unwrapped and placed in the `String` object `message` (line 81). When the `keyStruck` method ends, the `draw` call back method executes, and line 42 outputs the thrown error message to the top of the game board (Figure 10.16d). The next time a key is struck, the object `message` is set to the empty string on line 48, which causes the `draw` call back method to eliminate the message after a valid snowman motion.

```
1    import edu.sjcny.gpv1.*;
2    import java.awt.*;
3    import javax.swing.*;
4
5    public class ThrowingExceptions extends DrawableAdapter
6    { static ThrowingExceptions ge = new ThrowingExceptions();
7      static Game board gb = new Game board(ge, "THROWING EXCEPTIONS");
8      static BoxedSnowman2 s1;
9      static String message = "";
10
11     public static void main(String[] args)
12     { String s;
13       boolean correctXY = false;
14       int x, y;
15
```

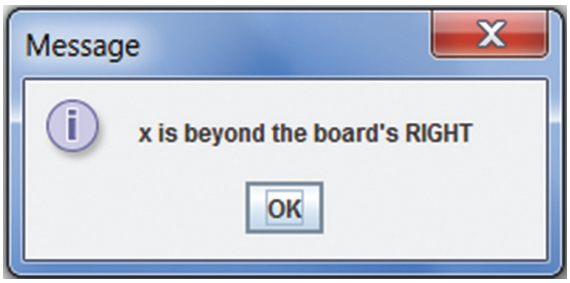
```
    while(correctXY == false) //x or y is not valid
    { s = JOptionPane.showInputDialog("enter the snowman's " +
                                      "X coordinate");
      x = Integer.parseInt(s);
      s = JOptionPane.showInputDialog("enter the snowman's " +
                                      "Y coordinate");
      y = Integer.parseInt(s);

      try
      {
        s1 = new BoxedSnowman2(x, y, Color.BLUE);//exception produced?
        correctXY = true;
      } //end try
      catch(RuntimeException e)
      {
        JOptionPane.showMessageDialog(null, e.getMessage());
      } //end catch
    } //end while

    showGame board(gb);
  }

  public void draw(Graphics g)
  {
    g.setColor(Color.BLACK);
    g.setFont(new Font("Arial", Font.BOLD, 18));
    g.drawString(message,  120, 50);
    s1.show(g);
  }

  public void keyStruck(char key)
  { int newX, newY;
    message = "";

    try
    {
      switch (key)
      {
        case 'L':
        {
          newX = s1.getX() - 2;
          s1.setX(newX); //could cause an exception
          break;
        }
        case 'R':
        {
          newX = s1.getX() + 2;
          s1.setX(newX); //could cause an exception
          break;
```

```
        }
        case 'U':
        {
          newY = s1.getY() - 2;
          s1.setY(newY); //could cause an exception
          break;
        }
        case 'D':
        {
          newY = s1.getY() + 2;
          s1.setY(newY); //could cause an exception
        }
      }
    } //end try
    catch(RuntimeException e)
    {
      message = e.getMessage();
    } //end catch
  }
}
```

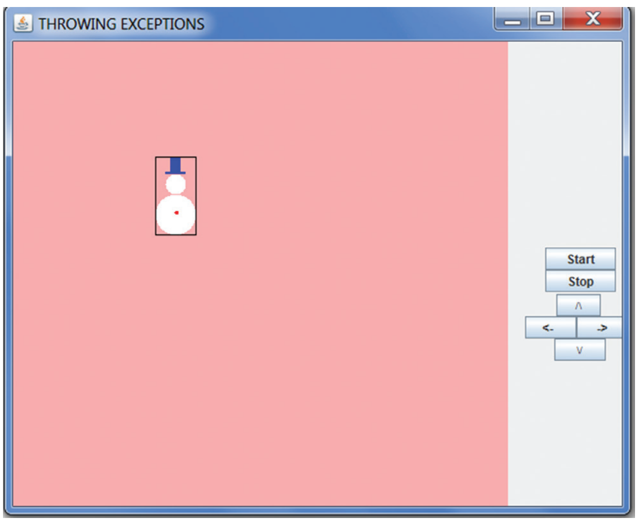
Figure 10.15
The application **`ThrowingExceptions`**.



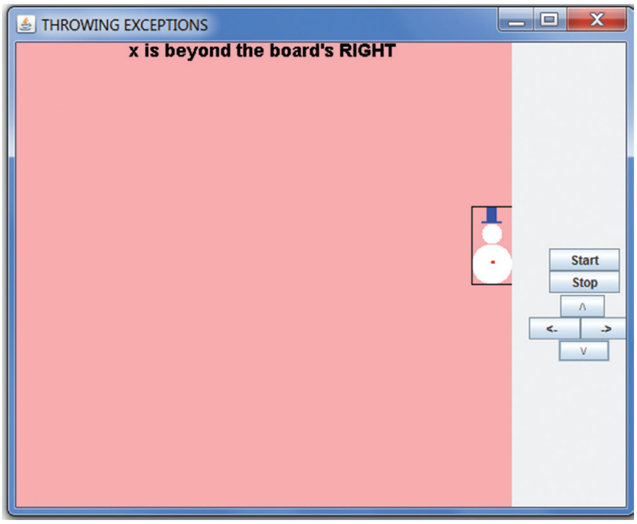
(a)



(b)



(c)



(d)

Figure 10.16
Output produced by the application **`ThrowingExceptions`**.

10.5 DEFINING EXCEPTION CLASSES

A new exception class can be defined by extending an existing exception class. Defining and using new exception classes in our programs makes them more readable and easier to understand because the names we give to them can be more representative of the error that caused the exception to be thrown. For example, if we defined an exception class named `OffBoardException` and used it on line 41 of Figure 10.14 and line 29 of Figure 10.15, the reason for throwing and catching the exception would be self-evident.

The concepts involved in and the syntax used to extend exception classes are the same concepts and syntax that apply to extending non-exception classes discussed in Chapter 8. The child class inherits all of the methods and data members in its inheritance chain, which includes the `getMessage` method defined in the class `Throwable`.

NOTE *The child of a checked exception class is always a checked exception class. The child of an unchecked exception class is always an unchecked exception class.*

Figure 10.17 presents the definition of the exception class `OffBoardException`, which is an exception class because it extends `RuntimeException` and is an unchecked exception class because `RuntimeException` is an unchecked exception class. The brevity of its code is typical of most non-API exception classes and can be used as a template for defining other exception classes. Both of its constructors simply invoke the `RuntimeException` class's constructor. The message passed to the class's one-parameter constructor (line 7) is passed to its parent's constructor, which wraps it around the object it creates. The inherited default message is the null string. This class could be used everywhere the `RuntimeException` class is used in Figures 10.14 and 10.15.

```
public class OffBoardException extends RuntimeException
{
  public OffBoardException()
  {
    super();
  }
  public OffBoardException(String message)
  {
    super(message);
  }
}
```

Figure 10.17
The non-API exception class **`OffBoardException`**.

Catch Block Ordering

The application `DefinedExceptionClass`, shown in Figure 10.18, demonstrates the order in which exception objects in the same inheritance chain must be caught, whether they are API classes or defined exception classes. The `catch` clause that catches an `OffBoardException` object (line 13) is coded before the clause that catches its parent class's `RuntimeException` object (line 19), which is coded before the clause that catches its parent class's `Exception` object (line 25).

The correct ordering of the `catch` blocks is up the inheritance chain of the exception objects from child to ancestors. This implies that when one of the `catch` blocks in a `try-catch` construct catches a `Throwable` exception object, that `catch` clause must be coded last. The ordering is a consequence of the polymorphic feature of inheritance. Because a parent type parameter can point to (reference) a child, a `catch` clause that catches an instance of a super class will also catch an exception object in a class that inherits directly or indirectly from it. Coding the `catch` blocks of a `try-catch` construct in any other order produces a translation error because the `catch` blocks that catch the child class exceptions have been rendered unreachable.

The method `test,` coded on lines 32–48 of Figure 10.18, throws exception objects when it is passed the value 1, 2, or 3. It is invoked on line 11 inside a `try` block that begins on line 9 and is passed the loop variable of the `for` loop that begins on line 7. This causes it to throw one of three different types of exceptions. Because one of the exceptions is a checked exception (an instance of the class `Exception` on line 36), the method's signature (line 32) contains a `throws` clause.

The order of the `catch` clauses coded on lines 13–29 is consistent with the inheritance chain of the exception classes `OffBoardException` (Figure 10.17), which inherits from the class `RuntimeException`, which inherits from the class `Exception` (Figure 10.1). These `catch` clauses output the message attached to the objects they catch, proceeded by annotation particular to each `catch` clause (Figure 10.19). The messages are created on lines 36, 40, and 45.

```
import java.io.IOException;

public class DefinedExceptionClass
{
  public static void main(String[] args)
  {
    for(int i=1; i<=3; i++)
    {
      try
      {
        test(i);
      }
      catch(OffBoardException e) //a child of RuntimeException
      {
        System.out.print("Caught by the OffBoardException " +
                         "catch block: ");
        System.out.println(e.getMessage());
      }
      catch(RuntimeException e) //a child of Exception
      {
        System.out.print("Caught by the RuntimeException " +
                         "catch block: ");
        System.out.println(e.getMessage());
      }
      catch(Exception e) //a child of throwable
      {
        System.out.print("Caught by the Exception catch block: ");
        System.out.println(e.getMessage());
      }
    }
  }
  public static void test(int path) throws Exception
  {
    if(path == 1) //throw an Exception object
    {
      throw new Exception("a message attached to an Exception object");
    }
    if(path == 2) //throw a RuntimeException object
    {
      throw new RuntimeException("a message attached to a " +
                                 "RuntimeException object");
    }
    if(path == 3) //throw an OffBoardException object
```

```
      {
        throw new OffBoardException("a message attached to an " +
                                    "OffBoardException object");
      }
    }
  }
```

Figure 10.18
The application **`DefinedExceptionClass`**.

```
Caught by the Exception catch block: a message attached to an Exception object
Caught by the RuntimeException catch block: a message attached to a RuntimeException object
Caught by the OffBoardException catch block: a message attached to an OffBoardException object
```

Figure 10.19
The output produced by the application **`DefinedExceptionClass`**.

10.6 CHAPTER SUMMARY

One very important programming goal is to produce reusable methods, and the concept of exceptions helps us achieve this goal. It gives us the ability to defer the decision as to what to do when an error occurs during the execution of a method to the invoker of the method. This extends the reusability of a method because the action to take when an error occurs is usually application dependent. One application may choose to terminate the program, while another application may re-invoke the method after giving the user a chance to correct an erroneous input.

Under Java's implementation of exceptions, the invocation of a method that could detect an error is coded inside the `try` clause of a `try-catch` construct, and the clause is immediately followed by one or more `catch` clauses. Each `catch` clause has a parameter list with one parameter whose type is the API class `Throwable` or one of its descendants. The type used in each parameter list must be different.

When the method detects an error, it executes a `throw` statement, which terminates the execution of the method and the invoker's `try` clause. The `throw` statement includes an exception object in the class `Throwable`, or one of its descendants, that contains information about the error. This object is passed to the `catch` clause that contains a parameter in the exception object's class or one of its ancestors. The code block of that, and only that, `catch` clause then executes to perform the application-dependent processing associated with this type of error.

Instances of the class `Throwable` and each of its subclasses contain default error information, which can be overwritten when these exception objects are created by passing a string to the classes' constructors. The code within the `catch` clauses can invoke the `getMessage` method on the exception object passed to them to fetch the string containing the exception error information.

The API classes `Exception` and `Error` are the two direct subclasses of `Throwable`. The class `Error` and all of its descendants are unchecked exception classes, while the `Throwable` and

`Exception` classes are checked exception classes. Conditions that cause instances of the `Error` class or its descendants to be thrown are considered abnormal and are best processed by the Java Runtime Environment. The method that contains the `try-catch` construct must catch a thrown checked exception, or the method must inform the translator that it will intentionally ignore the error via a `throws` clause added to its signature. If a thrown checked exception is not caught within an application, the runtime environment terminates the program and outputs the exception object's error information.

A `finally` clause can be used to implement tasks that should be performed after a `try-catch` block is completed. It is coded immediately after the last catch block, and it always executes. It must be coded after the `try` clause when there are no `catch` clauses associated with the `try` clause.

When designing an algorithm for a method, we should always consider what could go wrong and include a strategy in the algorithm to detect the error. Usually, the Boolean condition of an `if` statement is used to detect an error, and the `if` statement's code block creates and throws an exception object. A new exception class can be defined by extending an existing exception class. Defining and using new exception classes in our programs makes them more readable and easier to understand because the name we give to the class can be more representative of the error that caused the exception to be thrown. All of these design issues contribute to the development of readable, more reusable and maintainable software.

Knowledge Exercises

1. True or false:

a) An important programming design goal is to produce an error-free program.

b) Dividing a large program into smaller classes and testing each class guarantees that errors will not occur at runtime.

c) All uncaught exception objects are caught and handled by the Java Runtime environment.

d) The class `Throwable` has one direct subclass: `Exception`.

e) Abnormal conditions that cause an instance of the class `Error` to be thrown are best processed by the Java Runtime environment.

f) The class `Exception` is an unchecked exception class, and the class `Error is` a checked exception class.

g) An exception object is an instance of the class `Throwable` or one of its subclasses.

h) The API exception classes cannot be extended.

i) Any method that throws a checked exception must include a `throws` clause in its signature.

j) If a method throws an exception, the method invoking it must contain a `try-catch` block.

k) The code block of a `finally` clause always executes.

l) If the methods invoked inside a `try` block do not throw an exception, the program skips the `catch` block(s).

m) A `try-catch` construct can also be used to identify and process data as well as to detect errors.

n) The child of an unchecked exception class is always an unchecked exception class.
o) Some methods in the API classes throw exceptions.

2. Mention at least three things that might cause a runtime error or an exception in a program.
3. When would you choose to throw a checked rather than an unchecked exception?
4. Name a class you would extend to create an unchecked exception class.
5. Name a class you would extend to create a checked exception class.
6. Explain in some detail what happens when an error, such as division by zero, occurs in a Java program.
7. Tell whether each of these exceptions or errors is checked or unchecked:
   a) `IOException`
   b) `RunTimeException`
   c) `EOFException`
   d) `ArithmeticException`
   e) `NullPointerException`
   f) `AnnotationFormatError`
   g) `IndexOutOfBoundsException`
   h) `VirtualMachineError`
8. Explain the difference between the `throws` clause and the `throw` statement. Give an example of how each one is used.
9. Explain how you would fetch the string containing a caught exception object's error information.

Programming Exercises

1. Write a program that creates a three-element array and asks the user which element of the array should be output. Use a `try-catch` block to recover from an attempt to output an array element whose index is not 0, 1, or 2. Inform the user that an erroneous input was made, output the exception object's error information, and give the user an unlimited number of opportunities to correct the error. Do not use an `if` statement in this program.
2. Repeat Exercise 1 and include a separate method that is passed the array and the input index and performs the output. This method should use an `if` statement to detect an erroneous index and throw an unchecked exception containing the message: *The range of the index must be between 0 and 2.*
3. Repeat Exercise 2, modifying the method so it throws a checked exception in a new exception class named `InvalidIndexException`.
4. Repeat Exercise 2, modifying the method so it throws a checked exception in a new exception class named `IndexTooLowException` when the array index passed to the method is too low, and throws a checked exception in a new exception class named `IndexTooHighException` when the array index passed to the method is too high.

5. Write a program to output the number of operators contained within a valid arithmetic expression input by the program user. Use a `try-catch` construct to count the operators. Hint: the `parseDouble` method throws an exception when it is passed anything other than a string that represents a valid real number or a valid integer.

6. Write a static method named `inputInt` within a class named `ValidNumericInput` that displays the prompt passed to it in an input dialog box, accepts an integer input from the user, and returns the parsed integer. If the user does not enter a valid integer, or if the user clicks OK or Cancel without making an entry, the method throws an exception in the programmer-defined exception class `BadIntegerEntry`. The thrown exception object will contain a message indicating which event caused the exception: no entry or non-integer entry. Use the method in an application that gives the user an unlimited number of opportunities to enter two valid integers by invoking the method inside a `try-catch` construct coded inside a loop. Each time an erroneous input is made, the application will output the thrown exception object's error message before re-invoking the method. Note: a no-entry Cancel click returns `null`, and a no-entry OK click returns the empty string `""`.

7. Write a program to accept a sentence from the keyboard terminated by a new line. Use the `Integer` class's `parseInt` method to locate all the integers. Then, write them to the disk file *numbers.txt* (don't specify a path) one number per line in the order in which they appear in the sentence. After storing the integers in the disk file, ask the user which of the numbers to delete from the file via an invocation of the `inputInt` method described in Exercise 6. When the method throws an exception, the application should tell the program user which mistake (no-entry or non-integer entry) occurred and ask for another entry. Before terminating, the application should read the modified contents from the file and output them to the system console.

Enrichment

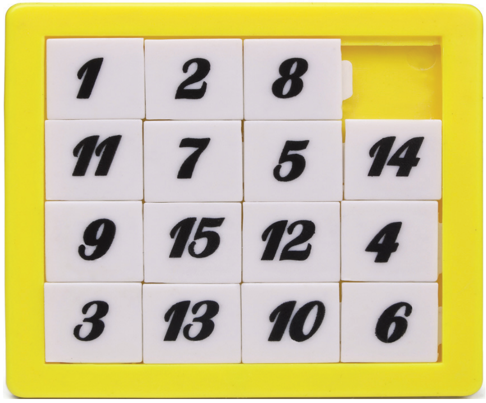
1. Read the API documentation on the class `Throwable` and its two direct subclasses.

2. Investigate how to determine from the API online documentation if an API method throws and exception and what exception it throws.

3. Investigate how the language C++ implements the concept of exceptions.

CHAPTER 11

GRAPHICAL USER INTERFACES



In this chapter

In this chapter, we will learn how to create more user-friendly and informative dialog boxes and how to build and incorporate graphical user interfaces (GUIs) into programs that can be run on computers as well as tablets and smartphones. The use of these point-and-click interfaces makes interacting with a program more user-friendly. Java provides three packages, the Abstract Window Toolkit (AWT), Swing, and FX to facilitate the development of dialog boxes and GUIs. FX will be used in most of this chapter.

Principles for designing a GUI interface will be explained and illustrated as will the techniques for adding GUI components to a program's window, and performing the processing associated with the user's interaction with its components. These components include panes, buttons, text fields, labels, tool tips, and 2-D graphics. Various layout managers, used to organize the components, will be discussed and compared.

Methods called event handlers will be discussed. These methods are invoked by the Java Runtime Environment (JRE™) when an event, such as a mouse click or a mouse drag, is performed on the GUI. We will learn how to write these event handler methods and how to register the methods with the Runtime Environment.

After successfully completing this chapter you should:

- Be able to create and use more informative and user-friendly dialog boxes.
- Know how to design and implement GUIs that contain panes, text fields, buttons, labels, tool tips and 2-D graphics using Java's FX features.
- Know the three (or sometimes four) step process for adding components to a container.

- Be familiar with layout managers to rapidly position components in a GUI window.
- Understand how to write event handler methods to process mouse, keyboard, timer, and paint events that occur on a GUI.
- Know techniques for registering event handler methods with the Java Runtime Environment, so they are executed when the specific events occur.
- Be familiar with Java Lambda expressions, and their role in registering event handlers.

11.1 OVERVIEW

A graphical user interface is a means of interacting with a program. Most often referred to using the acronym GUI (pronounced "goo-ee"), its design goal is to make the use of a program self-evident. GUIs are much more user-friendly than the original command-based interaction scheme in which a program would issue a text-based prompt that generically amounted to "what would you like to do?" and the user responded by typing a command such as "tax program".

Developed during the late 1970s, graphical interfaces were initially used to communicate with the operating system, but their power and ease of use was quickly adopted into all of the applications run on a system. Wherever possible, text-based prompts are replaced with icons, and keyboard input is replaced with mouse clicks, audio commands, and touch screen/pad input.

Just as graphical road signs succinctly communicate information to motorists, GUI objects, called *components,* permit us to quickly navigate our way through a program. The features of each of these components, which include clickable buttons, check boxes, radio buttons, scroll bars, sliders, and menu bars, to name a few, facilitate particular I/O functions that are common to most programs. Figure 11.1 shows some of the more commonly used components.

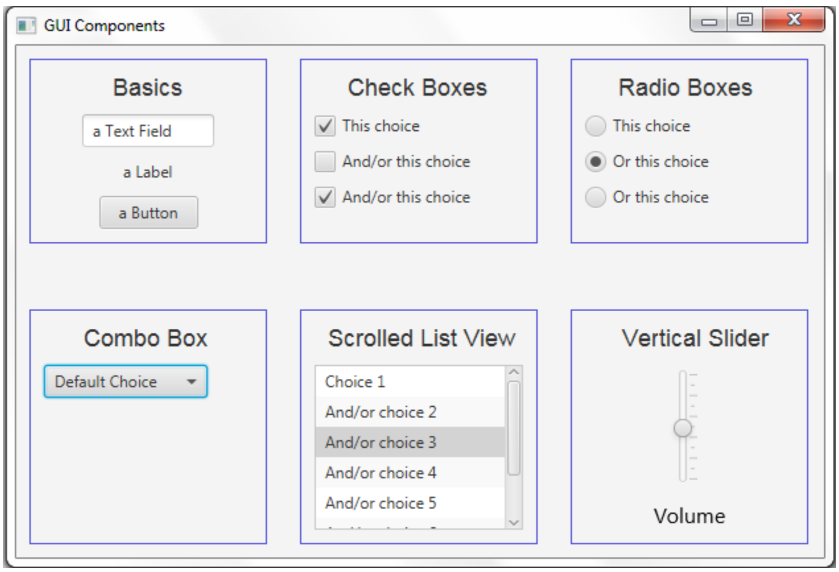


Figure 11.1
Commonly used GUI components.

While the use of GUIs has reduced the time and effort required to interact with a program, incorporating a GUI into a program can significantly increase the time and effort required to develop it. In reaction to this, many integrated development environments provide a GUI-builder feature that allows the programmer to rapidly develop the interface. It is built by selecting commonly used GUI components from a graphical display, dragging them to a position on a displayed window that will become the user interface, and then setting features associated with them such as their color, size, text type, and visibility. As the programmer builds the interface, the IDE adds the code to the program that creates and displays the components and adds empty methods to the program that will execute when the user interacts with the components. The programmer then adds the code to perform the application-dependent processing associated with the components to these methods.

Java's GUI Feature Evolution

The Java code to generate a graphical interface relies heavily on three groupings of classes added to Java. The first of these groupings of classes added was named the Abstract Windows Toolkit (AWT). This was superseded by the Swing collection of classes that both duplicated and extended the range of the types of GUI components available in the AWT package to include file and color-chooser dialog boxes. It also provided additional features such as tool tips and the ability to interact with the GUI in a drag-and-drop mode.

All Swing components were designed to be 100% cross-platform compatible. Java applications that use components in the Swing package are less dependent on the graphical features of the platform's operating system on which the application is run. For example, minor differences in the components' appearance (or look) and the change in their appearance when the user interacts with them (their feel) that are platform dependent can be eliminated. We already have some experience with using Swing GUIs, in that the input and message dialog boxes introduced in Chapter 2 are part of Swing's `JOptionPane` class.

In 2011, a third grouping of GUI generating classes named FX, was added to Java. It provides features not available in Swing, such as support for mobile touch devices, easier animation, and the use of special effects within the GUI, and enhanced ability to place GUI components inside of other GUI components. FX is the future of GUIs in Java. In future enhancements to Java GUIs will be added to FX. However, Oracle has stated that Swing will continue to be part of Java.

In the next section we will cover several techniques for enhancing the look and functionality of Swing's two dialog boxes. In the remainder this chapter, and in the entire next chapter, we will discuss the generation and use of FX graphical user interfaces. The equivalent topics for Swing graphical user interfaces are discussed in the Swing versions of these two chapters that are on the book's companion disk, and posted to the book's companion files FTP site.

11.2 ENHANCING SWING DIALOG BOXES

As we have learned in Chapter 2, Swing's dialog boxes provide the ability to easily add a graphical user interface to our programs to perform two tasks common to most applications: accept a user input in response to a displayed prompt; or to display a message to the user. In both cases the execution of the code that displayed the dialog box's window is suspended until the user responds to, or closes, the dialog box.

In addition to the versions of the `showInputDialog` and `showMessageDialog` methods discussed in Chapter 2, the `JOptionPane` class provides several overloaded versions of these methods and other methods that can be used to provide more informative and user-friendly dialog boxes. The default icon and title displayed in the dialog boxes can be changed, the dialog boxes can be displayed in the middle of a specified window such as the game board's window, and a default input or a set of input selections can be displayed in an input dialog box.

Table 11.1 presents a summary of the overloaded versions of the `showInputDialog` and `showMessageDialog` methods, with their signatures given in its left column. The check marks

Table 11.1
Options for Displaying Input and Message Dialog Boxes

Method	Feature(s) Incorporated into the Method				
	Default Input	Specify Window	Title	Icon	Input Choices
Input Dialog Boxes					
`showInputDialog(Object prompt)`					
`showInputDialog(Object prompt, Object defaultInput)`	√				
`showInputDialog(Component window, Object prompt)`		√			
`showInputDialog(Component window, Object prompt, Object defaultInput)`	√	√			
`showInputDialog(Component window, Object prompt, String title, int messageIcon)`		√	√	√	
`showInputDialog(Component window, Object prompt, String title, int messageIcon, Icon icon, Object[] selectionValues, Object initialSelectionValue)`	√	√	√	√	√
Message Dialog Boxes					
`showMessageDialog(Component window, Object message)`		√			
`showMessageDialog(Component window, Object message, String title, int messageIcon)`		√	√	√	

in the columns to the right identify the features of each version of the methods. The top section of the table presents the input dialog methods, and the message dialog methods are presented in the bottom section. The signatures of the methods that have been used up to this point in the textbook are shown at the beginning of the two sections of the table. All of the input dialog box methods return a reference to a `String` object except for the last one shown in the top portion of the table, which returns an `Object` reference.

The parameter `window` in the signatures of the message-box methods and last four input-box methods could be passed a reference to a GUI component such as a window. When it is, the dialog box is displayed in the center of the component. If the parameter is passed a `null` value, the dialog box is displayed in the center of the window of the program that invoked the method. To display it in the center of the game board window, the method would be passed the `GameBoard` object `gb` as shown on line 12 of the application `CenteredMsgBox`, shown in Figure 11.2. The output it produces is shown in Figure 11.3.

Normally, the `prompt` and `defaultInput` parameter used in the input dialog method signatures shown in the Table 11.1 are passed a `String` object. The `defaultInput` is displayed in the text area of the input box when it appears on the monitor, and can be changed (overtyped) by the program user. The argument passed to the parameter `title` (used in the fifth, sixth, and last rows of the table) is displayed in the title bar at the top of the dialog box. The parameter `message` in the message dialog method signatures is normally passed a `String` object or any object that contains a `toString` method. The parameter `icon` in the sixth row of the table is used to pass a programmer-defined instance of the `Icon` class to the method.

```
1   import edu.sjcny.gpv1.*;
2   import javax.swing.JOptionPane;
3
4   public class CenteredMsgBox extends DrawableAdapter
5   {
6     static CenteredMsgBox ge = new CenteredMsgBox();
7     static GameBoard gb = new GameBoard(ge, "My Game");
8
9     public static void main(String args[])
10    {
11      showGameBoard(gb);
12      JOptionPane.showMessageDialog(gb, "A Messages Box Centered " +
13                                        "in the Game Board Window");
14      showGameBoard(gb);
15    }
16  }
```

Figure 11.2
The application **`CenteredMsgBox`**.

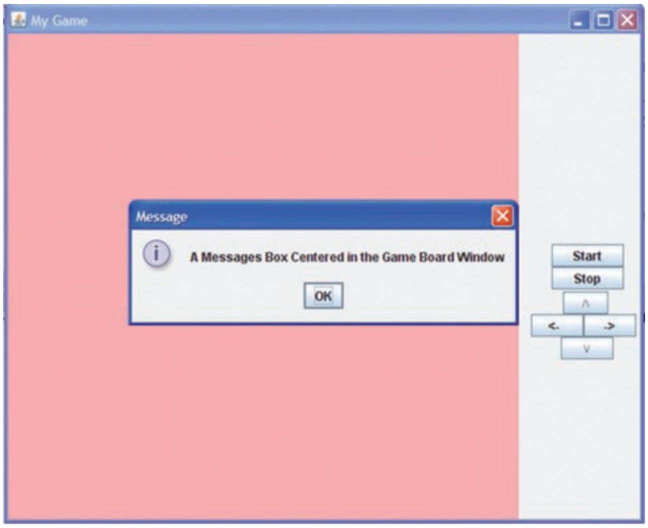


Figure 11.3
The output produced by the application **`CenteredMsgBox`**.

The methods in Table 11.1 whose signatures contain the parameter `messageIcon` can be passed any of five static constants defined in the class `JOptionPane`. This parameter is used to specify which one of five predefined icons will be displayed on the left side of a dialog box. Table 11.2 gives the names of the constants, their integer value, and the icons that are associated with each of them. The methods in Table 11.1 whose signatures do not contain the parameter `message-Icon` always display the default icons identified parenthetically in the rightmost column of Table 11.2. An integer literal between -1 and 3 inclusive (one of the five constants' values) can alternately be passed to this parameter.

Table 11.2
The **`JOptionPane`** Class's Predefined Dialog Box Icon Constants and Icons

Constant Name	Value	Icon	Common Icon Use
PLAIN_MESSAGE	-1	none	Other defined icons are inappropriate: no icon is displayed
ERROR_MESSAGE	0		An error or problem has occurred
INFORMATION_MESSAGE	1		For your information (message dialog box default)
WARNING_MESSAGE	2		Consider possible ramifications
QUESTION_MESSAGE	3		A reply to the prompt is requested (input dialog box default)

The last method shown in the input portion of the Table 11.1 implements all the features presented in that portion of the table. In this version of the method the default input is designated to be one of a valid set of inputs contained in an array passed to the method's `selectionValues` parameter. The designation of the default value is performed by passing one of the elements of the array to the parameter `initialSelectionValue`. The elements of the array can be `String` objects, instances of a class that contains a `toString` method, or several other options that will be discussed later in this chapter. If a `null` value is passed to the parameter `selectionValues`, the user can type the input value; otherwise, the user can only select one of the objects in the array, which are displayed in a drop-down box.

The application `EnhancedDialogBoxes` presented in Figure 11.4 demonstrates the use of all of the features implemented by the overloaded dialog box methods presented in Table 11.1, except centering the dialog box in a GUI component (which was demonstrated in the application presented in Figure 11.2). The dialog box outputs produced by the program are shown in Figure 11.5.

The string *ERROR* passed to the second parameter of the method invoked on line 12 of Figure 11.4 appears in the title area of the message box it outputs (Figure 11.5a). This message box also contains the non-default Error icon, whose number (0) is passed to the method's third parameter using the static constant `JOptionPane.ERROR_MESSAGE`.

Line 15 displays an input dialog box containing the default input, *Sophomore,* as shown in Figure 11.5b. The default value is passed to its second parameter on line 16.

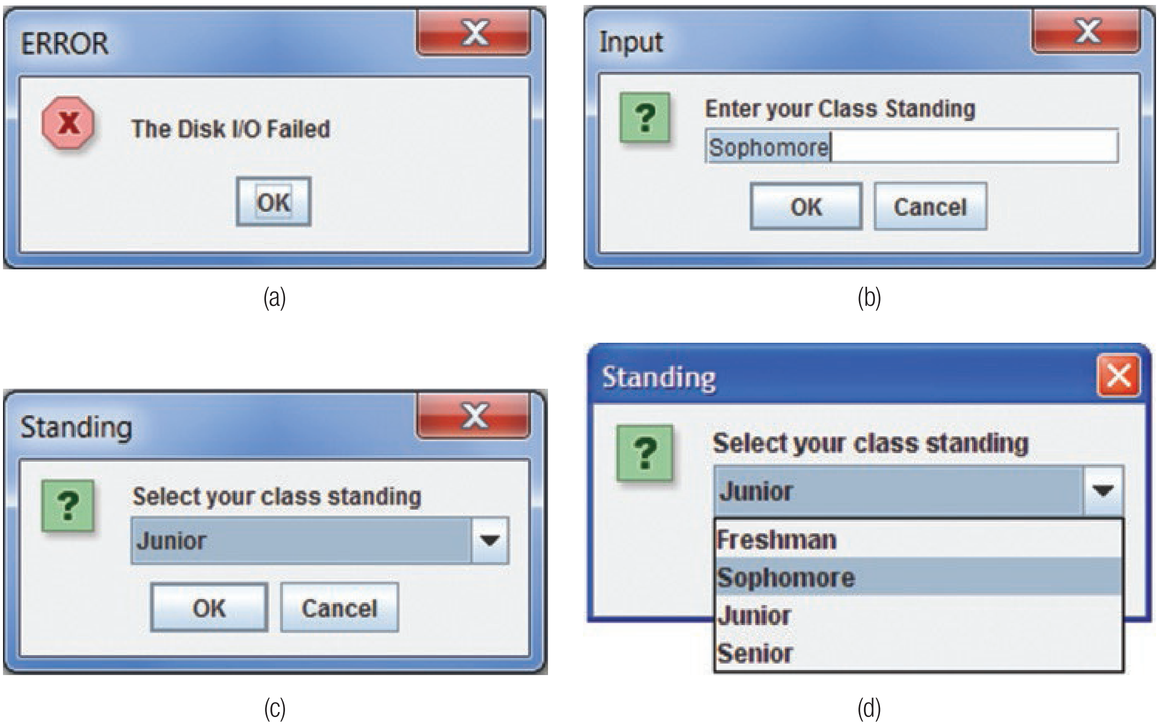
The method invoked on line 19 displays the input dialog box that is shown in Figure 11.5c. The box contains the title *Standing*, passed to the method's third parameter on line 21, and the Question icon because the numeric literal 3 is passed to its fourth parameter. The default input *Junior* is displayed in the text area of the input box because the third element of the array, defined on lines 7 and 8, is passed to the method's last parameter on line 24. The `null` value passed to the method on line 22 indicates that a programmer-defined icon is not passed to the method.

Figure 11.5d shows the input box displayed after the user clicks the box's down arrow to display the valid input choices passed to the method on line 23. The coercion on line 19 is necessary because the method invoked on that line returns an `Object` reference variable that contains the address of the user selected `String` object contained in the array passed to it on line 23.

```
1    import javax.swing.JOptionPane;
2
3    public class EnhancedDialogBoxes
4    {
5      public static void main(String[] args)
6      {
7        String[] inputOptions = {"Freshman", "Sophomore",
8                                 "Junior", "Senior" };
9        String s1, s2;
10
11       // Titled message box with an error icon
```

```
    JOptionPane.showMessageDialog(null, "The Disk I/O Failed", "ERROR",
                                  JOptionPane.ERROR_MESSAGE);
    // Input box with a default input
    s1 = JOptionPane.showInputDialog("Enter your Class Standing",
                                     "Sophomore");

    // A Non-default icon titled Input box, a valid set of inputs
    s2 = (String) JOptionPane.showInputDialog(null, "Select your " +
                                              "class standing",
                                              "Standing", 3,
                                              null,
                                              inputOptions,
                                              inputOptions[2]);

    System.out.println(s1 + " " + s2);
  }
}
```

Figure 11.4
The Application **EnhancedDialogBoxes**.



(a) (b) (c) (d)

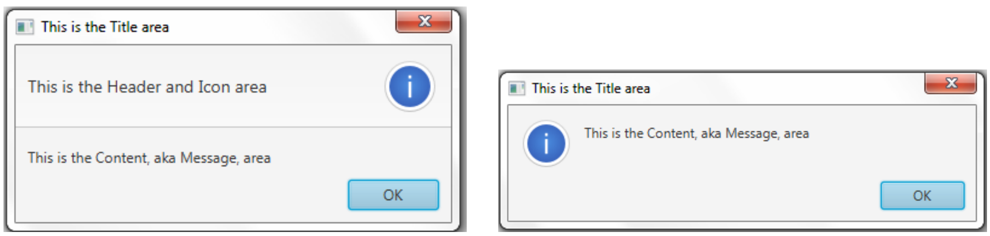
Figure 11.5
The output produced by the application **EnhancedDialogBoxes**.

11.3 FX DIALOG BOXES

The early versions of FX did not support dialog boxes, but current versions do. Not only do these dialog boxes provide all of the functionality and enhancements of Swing's dialog boxes described in Section 11.2, but FX's implementation of dialog boxes permits the programmer to add additional GUI components to them. For example, we can add a second input text field to one of FX's dialog boxes so that the user can input both a username and password before clicking the dialog box's OK button.

FX's `Alert`, `TextInput`, `ChoiceDialog`, and `Dialog` classes are used to create dialog boxes. An instance of the FX `Alert` class is analogous to a Swing message dialog box, and instance of the FX `TextInputDialog` class is analogous to a Swing input dialog box. An instance of the FX `ChoiceDialog` class is an input dialog box that can display a group of input choices. The FX class `Dialog` can be used to construct other programmer composed dialog boxes.

The graphical interface that represents instances of these classes share a common layout consisting of the three rectangular areas, sometimes referred to as bars, shown in Figure 11.6(a). The top area of the dialog box is referred to as its Title area, the middle area its Header area, and the bottom area its Content area.



(a) (b)

Figure 11.6
FX dialog box areas.

The Header area can be eliminated as shown in the dialog box depicted in Figure 11.6(b). When it is eliminated, the icon is placed in the Content area. This presentation of an FX dialog box more closely models the Swing dialog boxes that do not contain a Header area.

Unlike Swing dialog boxes, the code that creates an FX dialog box must end with a separate statement that causes the dialog box to be displayed to the user. Normally the method invoked is the method `showAndWait` on the dialog box instance. This not only displays the dialog box, but also pauses the program execution pending a user action on the dialog box. The action includes clicking a button displayed within the dialog box, clicking the X in its title bar, or striking the keyboard's Esc or Enter key. Assuming the dialog box was named `aDialog`, the line of code would be:

```
aDialog.showAndWait();
```

This line of code, and the code to create the dialog box that will be introduced in the next section, must be placed inside a JavaFX application. The structure of a JavaFX application is different from the structure of the Java (non-GUI) applications discussed thus far in this text. It will be explained in Section 11.3.2.

11.3.1 FX Message Dialog Boxes

The easiest way to create an FX message dialog box that displays a message to the program user is to declare an instance of the `Alert` class using the class's three-parameter constructor. To output the message "The Program is terminating" we would code:

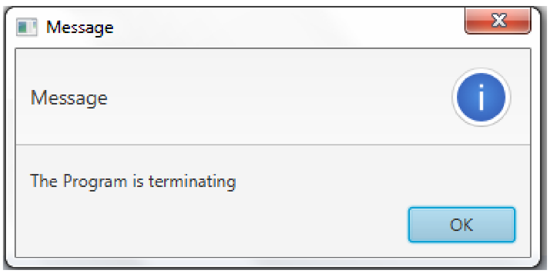
```
Alert messageDialog = new Alert(Alert.AlertType.INFORMATION,
                           "The Program is terminating", ButtonType.OK);
messageDialog.showAndWait();
```

This would produce the message box shown in Figure 11.7(a), and the program execution would pause pending a user action on the dialog box. The first argument passed to the constructor specifies the icon to be displayed, the second argument is the displayed message, and the third argument determines the annotation on the box's button. Three other convenient choices for specifying the displayed icon are the `AlertType` class's static constants `WARNING`, `ERROR`, `CONFIRMATION` that display the other three icons shown in Table 11.2. The `AlertType` class's static constant `NONE` is used to eliminate the display of an icon within a dialog box.

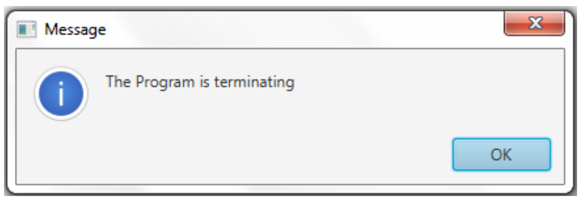
To eliminate the Header area of the dialog box and move the displayed icon to the Content area, we can declare an instance of the `Alert` class using the class's one parameter constructor. Then we would invoke two other `Alert` class methods to eliminate the Header area and specify the displayed message. To output the message "The Program is terminating" we would code:

```
Alert messageDialog = new Alert(Alert.AlertType.INFORMATION);
messageDialog.setHeaderText(null); //Eliminates the header
messageDialog.setContentText("The Program is terminating"); //Displayed message
messageDialog.showAndWait();
```

This would produce the message box shown in Figure 11.7(b). When the three-parameter constructor is used to create the `Alert` object, inserting the above invocation of the `setHeaderText` method in between the invocation of the constructor and the `showAndWait` method will also eliminate the Header area.



(a)



(b)

Figure 11.7
FX messages dialog boxes.

Normally, for a message dialog box, we would like the text displaed in the Title area to be the default text "Message". When that is not the case, we can use the `Alert` class's inhertited `setTitle` method to change the text. The below line of code would change the displayed title bar text of a dialog box named `dialogBox` to "Thought you should know".

```
dialogBox.setTitle("Thought you should know");
```

This statement can be used to change the title bar text of any type of an FX dialog box.

11.3.2 The FX Program Structure

The structure of a JavaFX GUI application, which is shown in Figure 11.8, does have two things in commom with the structure of the Java applications presented in previous chapters. They both begin by defining a program's class (line 6), and they both contain the method `main` that is invoked by the Java Runtime Environment to begin the program's execution (line 19).

However, in an FX applicaion, the program's class must extend the class `Application` (line 6), and most often the method `main` in an FX application does not do much. It simply invokes the inherited method `launch` (line 21) and passes `launch` the arguments passed to it by the Java Runtime Environment. Then the FX application's life time begins.

First the application's empty GUI user interface is created, which for a desktop application is a window, and the Java Runtime Environment invokes the application's `init` method, then its `start` method, and then its `stop` method. The signatures of these methods are defined in the class `Application`. The `Application` class's implementation of the `init` and `stop` methods do not contain any code, and can be overridden within the application to perform intialization tasks, before the GUI interface is displayed, and shutdown tasks after the GUI interface is closed. These two methods are not overridden in Figure 11.8.

```
1    import javafx.application.Application;
2    import javafx.scene.Scene;
3    import javafx.scene.layout.Pane;
4    import javafx.stage.Stage;
5
6    public class BasicProgramTemplate extends Application
7    {
8      @Override
9      public void start(Stage primaryStage)
10     {
11       Pane root = new Pane();
12       Scene scene = new Scene(root, 450, 300);
13
14       primaryStage.setTitle("An FX program window");
15       primaryStage.setScene(scene);
16       primaryStage.show();
17     }
18
19     public static void main(String[] args)
20     {
21         launch(args);
22     }
23   }
```

Figure 11.8
FX program structure.

The `Application` class's `start` method is abstract, and must be overridden within the application (lines 8–17). This method contains one parameter that is passed the user interface object, an instance of the class `Stage` (line 9), created by the Java Runtime Environment. If we ignore the application's intialization and shutdown tasks, this method, rather than the FX application's method `main`, should be thought of as the entry point of a JavaFX application. Since `main` does not have access to the user interface that is passed to the `start` method, the code that would have been placed in the method `main` in a non-GUI Java application is coded in the `start` method.

In the FX theater analogy jargon, an instance of the class `Stage` is refered to as the application's "stage" to which we can add a "scene". The stage is a desktop application's window. The scene will be displayed on the stage. It is an instance of a `Scene` object (line 12). When the scene is created it is passed an instance of a decendent of the class `Parent`, which is declared on line 11 to be an instance of the `Pane` class named `root`. The choice of this class is application-dependent. It can be thought of as a "container" object, because it normally contains a collection of GUI components (buttons, text fields, ….) to be displayed on the user interface. These objects, being application-dependent, would be created and then added to the container object prior to line 12.

The collection of GUI components is stored in a tree structure within the container object, which is added to the scene by passing it to the scene object's constructor (line 12). The three-parameter version of the constructor can also accept the width and height of the scene. Lines 14–16 add a title to the top of the user interface (the stage) on line 14, add the scene of components to the stage (line 15), and then display stage (line 16).

Figure 11.9 shows the relationship between an FX program's stage, scene, GUI component container object, and the application's GUI components. The `start` method is coded in reverse order (from the bottom to the top of Figure 11.9). The GUI components are created first (which will be discussed later in this chapter), then the container object is created (line 11 of Figure 11.8) and the componets are added to it, then the container is added to the scene (line 12 of Figure 11.8), and then scene is added to the stage (line 15 of Figure 11.8).

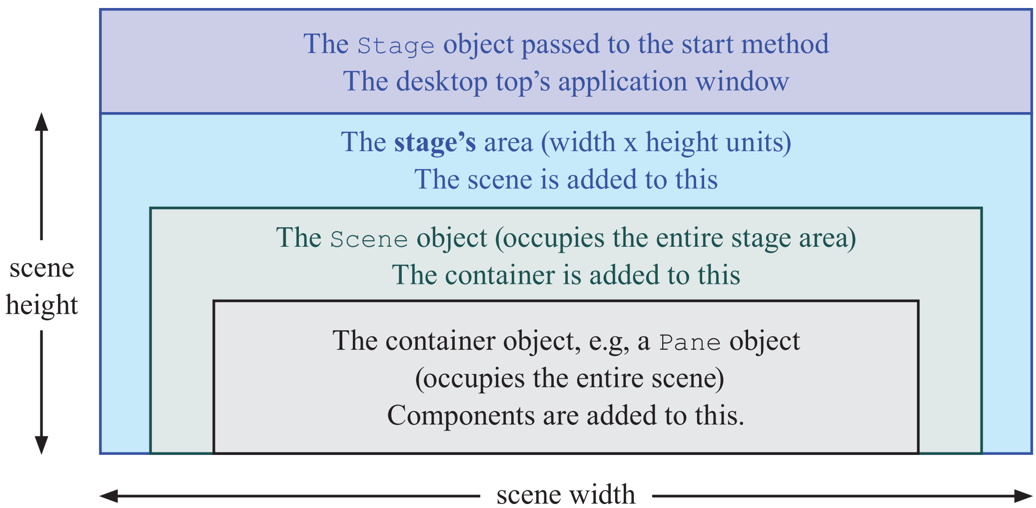


Figure 11.9
Relationship between and FX stage, scene, container, and GUI components.

Having gained an understanding of the FX program template, we can use it to write our first FX GUI application. By adding the below code to the end of the start method shown in Figure 11.8, the program's window will be displayed with an FX message dialog box horizontally centered on it as show in Figure 11.10.

```
Alert messageDialog = new Alert(Alert.AlertType.INFORMATION);
messageDialog.setHeaderText(null);
messageDialog.setContentText ("Close the program window" +
                              "\nto terminate the program");
messageDialog.showAndWait();
```

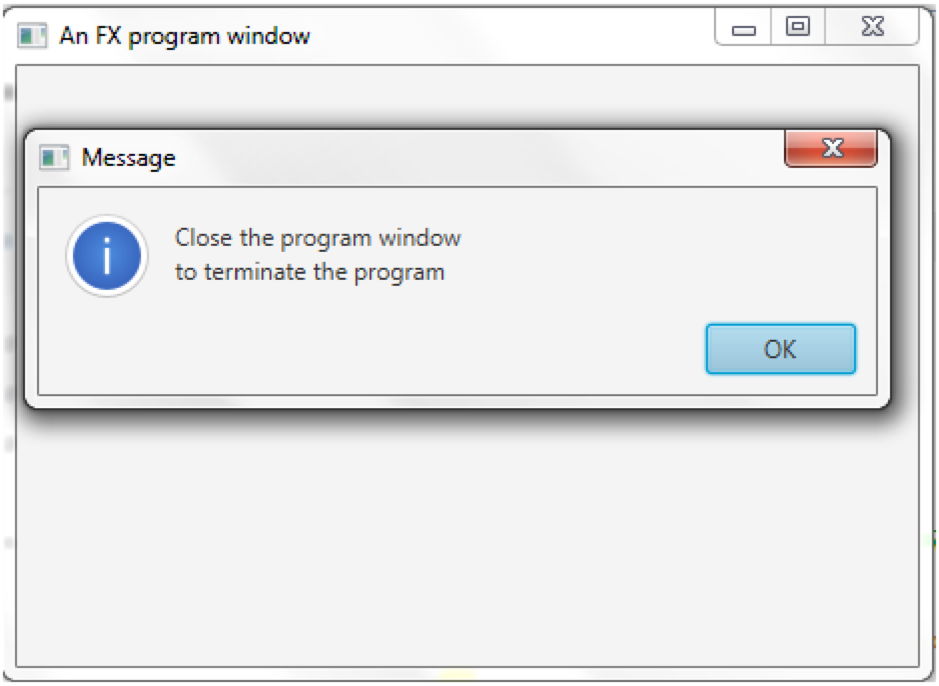


Figure 11.10
Display of a message box within an FX GUI application.

11.3.3 FX Input Dialog Boxes

Input dialog boxes are used to obtain input from the user, perhaps the user's first name, in response to a displayed prompt. To create an FX input dialog box we declare an instance of the `TextInputDialog` class. This class contains a no-parameter constructor and a one-parameter constructor. The one-parameter constructor is used when we want display a default input to the user. To create an input dialog box, containing the prompt "Enter your name:" with a default input of "Mary", similar to the one shown in Figure 11.5(b) that does not contain a Header area, we would code:

```
TextInputDialog inputDialog = new TextInputDialog("Mary"); //Default input Mary
inputDialog.setTitle("Input dialog box");
inputDialog.setHeaderText(null); //Eliminates the header area
inputDialog.setContentText("Enter your name:"); //The prompt
inputDialog.showAndWait();
```

As discussed in Section 11.3.3, the invocation of the method `showAndWait` would pause the program execution pending a user action on the dialog box. However, when the dialog box is an instance of the class `TextInputDialog`, we also need to obtain the contents of the dialog box's text box before execution continues.

To do this we simply modify the invocation of the `showAndWait` method in the above code segment to capture the address of the returned instance of the class `Optional`. In our discussion of FX message boxes we ignored the returned object, which is always an option in Java. The code to place the address of the returned object in the variable `response` is:

```
Optional<String> response = inputDialog.showAndWait();
```

The inclusion of the characters `<String>` in the above statement causes the translator to check that the instance of the class `Optional` returned from `showAndWait` contains a `String` data member. Its inclusion here is not necessary, but it is considered good programming practice as will be discussed in Chapter 13 Generics. To fetch the string stored in the returned object's data member we invoke the `get` method on the returned object. The below statements display the input dialog box `inputDialog` and then output the user input to the system console.

```
Optional response = inputDialog.showAndWait();
System.out.println("Your name: " + response.get());
```

The `isPresent` method can be invoked to determine if the user clicked the dialog box's title bar's X or the Cancel button to close out the dialog box. Either of these actions would cause it to return false.

11.3.4 A Complete FX Application

Figure 11.11 is a complete FX application that uses the FX program structure (shown in Figure 11.8) to create and display the application's GUI window, and uses the dialog box skills we have learned to interact with the user. It also introduces the use of the `Optional` class's `isPresent` method. The application's window is not displayed until the user enters his/her name via a displayed input dialog box. The user's name is then incorporated into the window's title, and the window is displayed. The program's input dialog box displaying the default user name Mary is shown in Figure 11.12(a), and the application's window displayed after the user entered Alice via the dialog box is shown in Figure 11.12(b).

Lines 8–12 and lines 26–39 of the FX application's code shown in Figure 11.11 are part of the FX program structure. When the `start` method (line 11) begins execution, lines 14–17 create an input dialog box with no Heading area (line 16).

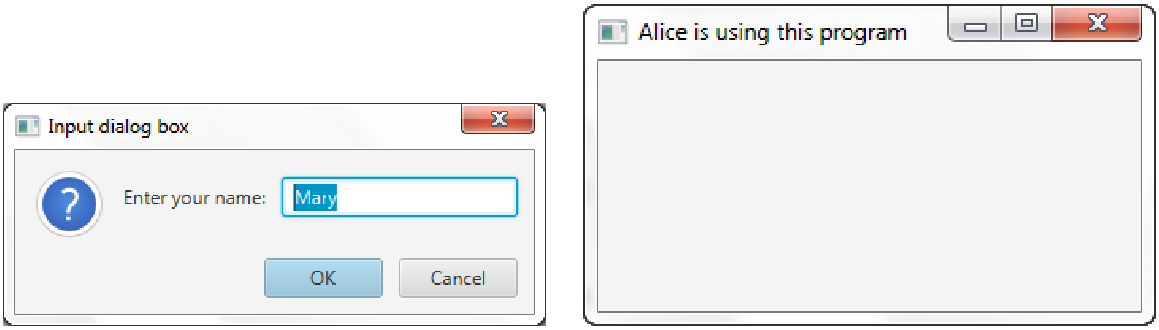
Then on line 21, the dialog box is displayed and the returned object is stored in the variable `response`. This statement is coded inside a `do-while` loop that begins on line 19, to force the user to either accept the default name Mary, or enter a different name. The Boolean condition of the loop on line 22 contains an invocation of the method `isPresent` on the returned object response. The method returns `false` if the user clicked the dialog box's Cancel button or the X in its title bar, i.e. a response was not made. This causes another iteration of the loop to be performed, and the dialog box is redisplayed.

When the loop ends, line 24 invokes the `get` method on the object `response` to fetch the string containing the entered name, and output it to the system console. A similar invocation is used on line 29 to add the user's name to the application window's title bar, and then line 32 displays the application's window.

```
1    import java.util.Optional;
2    import javafx.application.Application;
3    import javafx.scene.Scene;
4    import javafx.scene.control.TextInputDialog;
5    import javafx.scene.layout.StackPane;
6    import javafx.stage.Stage;
7
8    public class FirstFXApplication extends Application
9    {
10     @Override
11     public void start(Stage primaryStage)
12     {
13       Optional response;
14       TextInputDialog inputDialog = new TextInputDialog("Mary");
15       inputDialog.setTitle("Input dialog box");
16       inputDialog.setHeaderText(null);
17       inputDialog.setContentText("Enter your name:");
18
19       do
20       {
21         response = inputDialog.showAndWait();
22       } while (response.isPresent() == false);//no name entered
23
24         System.out.println("The name entered is " + response.get());
25
26         StackPane root = new StackPane();
27         Scene scene = new Scene(root, 450, 300);
28
29         primaryStage.setTitle(response.get() +
30                                  " is using this program");
31         primaryStage.setScene(scene);
32         primaryStage.show();
33     }
34
```

```java
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 11.11
The application `FirstFXApplication`.



(a) (b)

Figure 11.12
The application `FirstFXApplication` dialog box and window.

11.3.5 FX Choice Dialog Boxes

A choice dialog box is used to obtain an input the user selects from a group of displayed valid inputs. For example, a user is asked to input his/her class standing, and the choices are Freshman, Sophomore, Junior, or Senior.

To create an FX choice dialog box we declare an instance of the `ChoiceDialog` class. This class contains a no-parameter constructor, which is not often used. There are two other constructors that both use the first argument passed to them as the default input choice. The most often used of these two constructors considers the second argument passed to it to be an array containing the valid choices. The other constructor accepts an unlimited number of arguments, with the first one being the default choice, and the remaining arguments being the other valid choices.

To create a choice dialog box named `inputDialog` with no Header area, similar to the one shown in Figure 11.5(d) but with a default input of "Freshman", we would code:

```java
String[] choices = {"Freshman", "Sophomore", "Junior", "Senior"};
ChoiceDialog dialog = new ChoiceDialog ("Freshman", choices);
dialog.setTitle("Choice Dialog");
dialog.setHeaderText(null);
dialog.setContentText("Select your class standing");
```

Then, as we have discussed in Section 11.3.3, the code to pause the program's execution pending a user action on the dialog box, and place the user's selection in the `String` variable `status` we would code:

```
Optional response = dialog.showAndWait();
if (response.isPresent())
{
  String status = response.get();
}
```

The FX application `ChoiceDialogBoxes` shown in Figure 11.13 uses these coding techniques to solicit a student's status using the dialog box shown in Figure 11.14(a), and then displays the application's window with the student's status incorporated into the window's title, as shown in Figure 11.14(b).

It uses a different coding style than that used in the application presented in Figure 11.11. The code to create the dialog box (lines 25–29) and the code to display it and capture the returned `Optional` object (line 31), are coded inside a method named `getStanding` that begins on line 23. From a software engineering viewpoint, this divide-and-conquer coding style is preferred. It makes the code more readable, and the method could be written and tested by one programmer, with the rest of the program written by another programmer.

The method is a non-void method. On line 34 it returns the `Optional` object produced by `showAndWait` to the invoker if the user selected one of the input choices (i.e., did not click the X in the dialog box's title). As we have previously discussed, the Boolean condition on line 32 is true when that is the case. Otherwise, on line 37, the method returns the `Optional` object returned from the invocation of the `Optional` class's static method `ofNullable` that contains the empty string passed to it.

The code `getStanding().get()`on line 17 of the start method invokes the `getStanding` method, and then the `Optional` class's `get` method is invoked on the returned object to fetch the returned string. Then that string is concatenated to the rest of the window's title annotation. Line 17 is a succinct way of coding the following two lines:

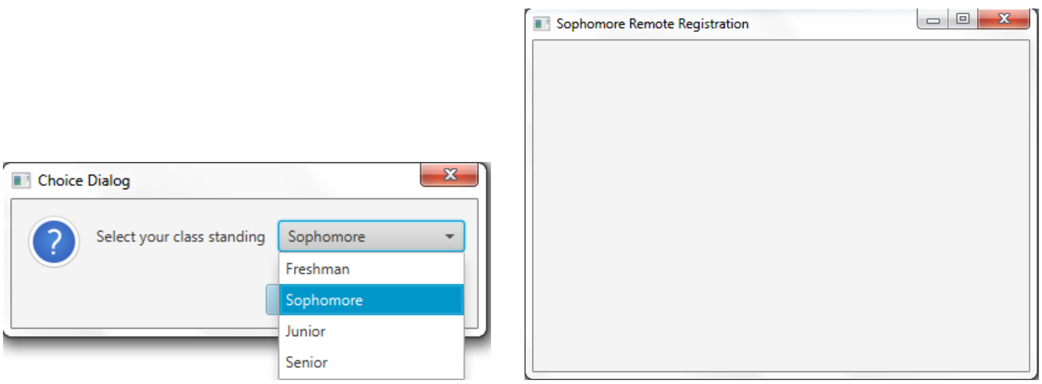
```
Optional optionalObject = getStanding();
primaryStage.setTitle(optionalObject.get() + " Remote Registration");
```

Also in the interest of brevity, the number of import statements (lines 1–7) included in the program has been reduced by replacing the names of the imported classes that would appear after the last period in the import statements, with an asterisk. For example, if the application used three classes in the `util` package, the three import statements used to import them have been replaced with line 1.

```
1   import java.util.*;
2   import javafx.application.*;
3   import javafx.event.*;
4   import javafx.scene.*;
5   import javafx.scene.control.*;
6   import javafx.scene.layout.*;
7   import javafx.stage.*;
8
9   public class ChoiceDialogBoxes extends Application
10  {
11    @Override
12    public void start(Stage primaryStage)
```

```
    {
        StackPane root = new StackPane();
        Scene scene = new Scene(root, 450, 300);

        primaryStage.setTitle(getStanding().get() +" Remote Registration");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private static Optional getStanding()
    {
        String[] choices = {"Freshman", "Sophomore", "Junior", "Senior"};
        ChoiceDialog dialog = new ChoiceDialog ("Freshman", choices);
        dialog.setTitle("Choice Dialog");
        dialog.setHeaderText(null);
        dialog.setContentText("Select your class standing");

        Optional response = dialog.showAndWait();
        if (response.isPresent())
        {
            return response;
        }
        else
        {   return Optional.ofNullable("");
        }
    }
    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 11.13
The application `ChoiceDialogBoxes`.



(a) (b)

Figure 11.14
The application `ChoiceDialogBoxes`'s dialog box and window.

11.4 CREATING A GUI INTERFACE

Dialog boxes are a convenient way of outputting messages to the user and obtaining an input from the user during the execution of a program. However, most of the user interactions with an application are with GUI components contained in the program's window such as buttons, text boxes, radio buttons, and other components.

JavaFX provides a rich assortment of GUI components, some of which are shown in Figure 11.1, that can be added to a window. While there is some overlap in the roles that they play in the I/O process, each component has been designed to facilitate a particular I/O function. For example, the functionality of radio buttons makes them the best components to use to acquire one choice from a small set of mutually exclusive choices.

The most common components used in GUI interfaces are buttons, text fields, labels, check boxes, radio buttons, combo boxes, and list views. Table 11.3 contains the constructor methods commonly used to create these components, grouped by the components' intended functionality, and gives a brief description of the I/O function they were designed to facilitate. It also includes the names of the container classes that are most commonly used to collect and position these components.

Table 11.3
Commonly Used JavaFX GUI Components

Commonly Used Component Constructors **Input Components**	Component's Targeted Use
`CheckBox(String text)`	Select one or more inputs from a small group of suggested inputs by clicking one or more boxes
`RadioButton(String text)`	Select one input from a small group of suggested inputs by clicking a button
`ComboBox(ObservableList itemsDisplayed)` `ListView(ObservableList itemsDisplayed)`	Select one input (`ComboBox`) or several (`ListView`) from a large group of suggested inputs
Input or Output Component	
`TextField(String text)`	Keyboard input, String output
Annotation or Output Component	
`Label(String text)`	Annotate a window including placing prompts at text boxes; String output to the window
Initiate Processing Component	
`Button(String text)`	Execute instructions associated with the click of the button
Containers: Collect and Position Related Components and 2D Graphics Components	

`Pane(), Group()`
`HBox, VBox, StackPane(), GridPane(),`
`BorderPane, FlowPane, ...`
	Group components and 2D shapes, and position them in the window

The components listed in the first six rows of Table 11.4 are often referred to as controls, because they are descendents of the `Control` class and the user interacts with them to control the execution path of the program. The container classes listed in the bottom row of the table are used to group *and* position components added to them. All of them except the Group class are descendents of the `Pane` class. That is why some of their names end with the word Pane. Instances of these classes not only can be added to a scene, but they can also be added to other container objects.

11.4.1 Designing the Interface

Before beginning the coding process of declaring and adding GUI components to the user interface, it is very useful to make a quick sketch of the interface's design, which shows all of the components to be added and their position (layout) in the window. A design sketch of an adding machine GUI is shown in Figure 11.15. Its level of detail is typical of that contained in a design sketch.

The choice of which components to add is based on the I/O requirements of the program and the component's targeted use listed in Table 11.3. As noted in the table, the `TextField` component can be used for both input and output.

If the program involves a series of user interactions that should be entered in a particular order, adding input components to the window from left to right and top to bottom in the order of interaction enhances the friendliness of the interface. The layout of the components in Figure 11.15 implies the user should enter the two numbers to be added, and then click the Compute button, before clicking the Clear button.

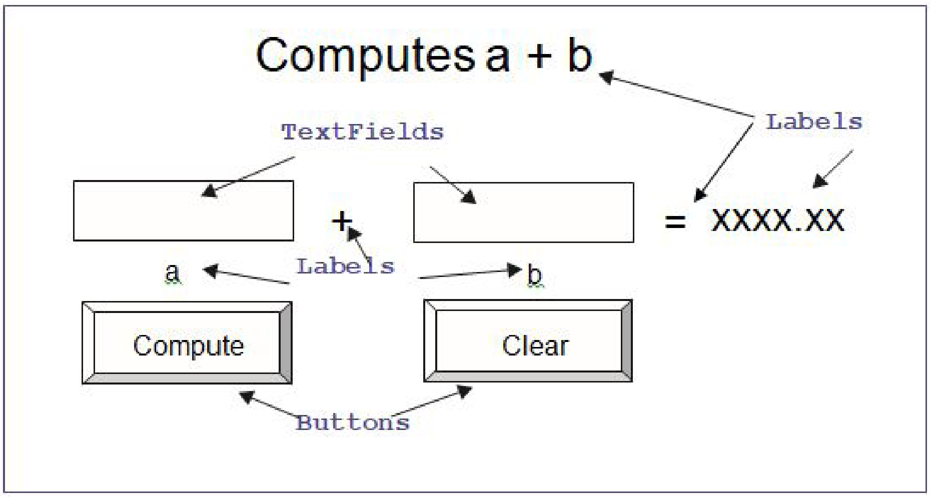


Figure 11.15
The GUI design for an adding machine.

After the design of the interface has been completed a four-step coding process is used to add the components to a container object:

1. Create the component control objects included in the design
2. Specify the components' properties such as size, font style, tool tip, visibility, and location
3. Create the container object and set its properties
4. Add the components to the container

11.4.2 Creating Controls and Setting Their Properties

The more commonly used constructors used in Step 1 of this process to create the components, are listed and described in Table 11.3. The string passed to these constructors and the observable list passed to the combo box's and list view's constructor, become the annotation that will appear on, within, or next to the component. For example, the following statement creates a button with the text *Click Me* displayed on it, and a text field with the text *Hamburger* displayed in it.

```
Button aButton = new Button("Click Me");
TextField entree = new TextField ("Hamburger");
```

Technically speaking, these constructors use the arguments passed to them to set properties of the object, after they create it. For example, a width property of a button and label component are set to accommodate the display of all characters in the string passed to it. In so doing they performed a portion of Step 2 of our process.

The top portion of Table 11.4 describes some of the methods used in Step 2 to specify properties of the components. The last row of the table gives a generic description of the method used to add a component to a container. As stated in that row, the type of the object it is invoked on and its parameter list depend on the container to which the component is being added. We will see an example of its use in the next section.

Table 11.4
Methods Used to Specify a Component's Properties and Add it to a Container

Method Signature Methods Invoked on Components	Description
`setText(String theText)`	Changes the text displayed on the component to `theText`
`String getText()`	Returns the text displayed on the component
`setFont(Font fontStyle)`	Sets the font style of the container or component that invoked the method to `fontStyle`
`setVisible(boolean visible)`	The component is made visible when the method is passed `true`, invisible when passed `false`
`setMinWidth(int mWidth)`	Sets the minimum width of a component to `mWidth`
`setMinHeight(int mHeight)`	Sets the minimum height of a component to `mHeight`

(Contd.)

Method Signature Methods Invoked on Components	Description
`setStyle("-fx-background-color: pink")`	Sets the background color of a container to pink
`setLayoutX(double x)` `setLayoutY(double y)`	Sets the component's location within an instance of a `Pane` to (`x`, `y`)
Methods to Add Components to Containers	
`add(ParameterList)` `addAll(ParameterList)`	Adds a component (or components) to a container. The type of the object they are invoked on, and the parameter lists, depend on the container

11.4.3 Adding the Controls to a `GridPane` Container

As discussed in Section 11.3.2, FX GUI control components are not added to a window or a scene. Rather they are added to a container object that is an instance of the class `Group`, or the class `Pane`, or a class that inherits from `Pane`. Then the container object is added to the scene object, which is then added to the stage object.

The two FX programs presented in this chapter so far declared an empty container instance of the class `Pane` or `StackPane` and added that object to the Scene object, as shown on lines 14 and 15 of the application presented in Figure 11.13. One way of adding controls to a `StackPane` container is to pass them to its constructor. The code to create and add two buttons to the center of the application's window would be:

```
// create the controls
Button button1 = new Button("Button 1");
Button button2 = new Button("Button 2");

// add the controls to container
StackPane thePane = new StackPane(button1, button2);
// add the pane to the Scene
Scene scene = new Scene(thePane, 300, 250);
```

The `StackPane` container is listed at the bottom of Table 11.3 as one of the FX containers that both contain *and* locate the components. This set of containers is referred to in the Java literature as "layout managers". In the case of a `StackPane` controls are displayed stacked on top of each other, and so the last control to be added to it, `button2` in the above code, would be the only one visible. If `button1` were larger than `button2`, we would see the portion of `button1` that `button2` did not cover. In some applications this is exactly what we want to happen, but it is clearly not a suitable container for our adding machine application.

The GridPane Layout Manager

The fact that our application's GUI interface (shown in Figure 11.15) consists of components arranged on four rows, and some of the rows (the bottom three rows) have some of their components in the same columns, suggests that we should use a `GridPane` container to position the components for us. Details about the use of other layout managers, and the way to determine the best layout manager for an application, will be discussed in Section 11.6.

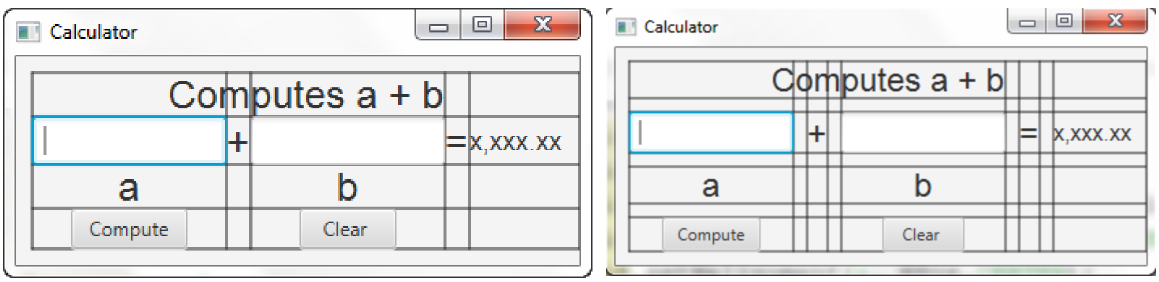
A `GridPane` container consists of a rectangular grid cells that components are placed and displayed in. The display of a component can extend to the set of adjacent cells below and to its right, if they are unoccupied. The row and column numbers of the cells begin at zero, and increase as we move down and to the right. When a component is added to the grid, its cell column and row number is specified. To permit the component to be displayed in its adjacent cells, the number of columns and the number of rows to display the component in is specified to be greater than one. Components added to the same cells are stacked on top of each other.

The number of rows and columns the grid will contain is not explicitly specified by the programmer. When container is constructed, it has no rows and columns. As components are added to the container, one row and/or one column is added to the grid if the component's row and/or column do not already exist, subject to the condition that every row and column must contain at least one component. Therefore, when all the components have been added to the grid, the number of rows and columns is equal the number of components that were added to a new row and the number of components that were added to a new column respectively.

The width of a column is set to the maximum of the widths of the components in that column, and the height of a row is set to the maximum of the heights of the components in that row. For example, the height of the two text fields in Figure 11.15 dictates the height of the row they and three labels occupy.

Assuming all the components in the adding machine's window occupy one column and one row, except for the label in row 0 that occupies five columns, Figure 11.16(a) shows the user interface with the column and row boundaries displayed. From the figure it is easy to see that the width of the columns is not the same. The height of all of the rows is also not the same, but that is not as easy to see.

The text fields in columns 0 and 2 of row 1, set the width of those columns and the height of that row. The width of the labels in columns 1, 3, and 4 set the width of those columns.



(a) (b)

Figure 11.16
An application's GUI column width and row heights, and the spacing between its columns and rows.

A GridPane's Properties

Table 11.5 lists the signatures and descriptions of methods used to set the properties of a `GridPane` object, and add components to it. To add spacing between the row and column boundaries, the `setHGap` and `setVGap` methods can be invoked on the `GridePane` object and passed the number of units of spacing to be inserted between the cell columns and cell rows. Figure 11.16(b) shows same user interface depicted Figure 11.16(a) with ten units of spacing inserted between the row and column boundaries. Assuming the name of the `GridPane` object is named `grid`, the code to accomplish this would be:

```
grid.setHgap(10);
grid.setVgap(10);
```

Table 11.5
Methods Invoked on a `GridPane` Object to Set its Properties and Add a Component to it

Method Signature	Description
`setHalignment(Node control, HPos HPos.CENTER)` `setValignment(Node control, VPos VPosVPos.CENTER)`	Centers the component `control` horizontally within its cell range Centers the component `control` vertically within its cell range
`Insets i = new Insets(int top, int right, int bottom, int left))` `setPadding(Insets i)`	Sets the spacing around the outside of the edges of a `GridPane` object
`setHgap(double value);` `setVgap(double value);`	Sets the horizontal and vertical spacing between cells
`setGridLinesVisible(boolean show)`	Shows the row and column edges, and the edges of the spacing between them when the argument passed to `show` is `true`
`add(Node aNode, int col, int row, int cSpan, int rSpan)`	Adds component `aNode` into the cell (`col, row`) and permits it to span `cSpan` columns and `rSpan` rows

Adding Components to a GridPane

After declaring the components and setting their properties, the components that make up the GUI interface design can be added to a `GridPane` container in any order. However, from a readability and debugging point of view, it's best to create and add them one row at a time starting at row 0, and working our way across the rows from left to right. Using this technique the code to declare the component in row 0 of our adding machine, set its text property, center it, and add it to a `GridPane` container named `grid` is:

```
Label description = new Label("Computes a + b");
grid.setHalignment(description, HPos.CENTER);
grid.add(description, 0, 0, 5, 1);
```

The third line of the above code places the label `description` in column 0 and row 0, and indicates when it is displayed it can occupy five columns and one row and should be centered in these columns. However, until we add other components to the interface to establish more columns, it would be displayed in column 0.

The application `AddingMachine` show in Figure 11.17 constructs the GUI interface shown in Figure 11.16 using a `GridPane` layout manager and then displays it. The code of the application follows our four-step coding process to create the user interface. Within each of these steps the coding style previously discussed, starting at row 0 of our interface design and working our way across the rows from left to right, is used.

Rather than declaring the reference variables used to store the address of the GUI components inside the `start` method, they are declared as class level variables on lines 12–14. The reason for this will be discussed in Section 11.5.

Beginning with row 0 of our interface design, the GUI components are created and their properties are set (Steps 1 and 2 of our coding process) on lines 20–39. These two steps are combined for each component in that a component is created and its properties are set before the next component is created. In some cases a property is set by the constructor (e.g., line 20), and in other cases a method listed in Table 11.6 is used (e.g. line 21).

Within lines 20–39 the `setFont` method is invoked several times to overwrite the default values of the font type and size properties of GUI components. It is passed a nameless instance of the `Font` class to specify the values of these two properties. Line 23 and 27 invoke the `setPrefSize` method to reduce the default size of the two text fields. Similarly, line 32 uses the `setMinWidth` method on the label sum, which sets the width of column 4, to ensure that that column is wide enough to display the number 9,999.99. Line 39 associates a tool tip with the `clear` button.

Step 3 of our coding process—create the container object and set its properties—is performed on lines 42–50. The `GridLayout` layout container `grid` is declared on line 42, and then the `GridLayout` class's methods presented in Table 11.5 are used to set some of the container's properties. The spacing between the cells is set on lines 43–44, and the spacing between the container and the edges of the window, is set on line 45. On this line, the right edge window spacing is set to 0, because the width of the label `sum`, set on line 32 provides sufficient spacing.

Lines 46–50 set the alignment property of the cells that five of the components will be displayed in, to center the components within those cells. The default is left justified. This is not necessary for the cells of the other five components, because these components are each displayed in one column *and* they set the width of that column. For example, the text field component `aValue` sets the width of column 0 and the label component `plus` sets the width of column 1, as shown in Figure 11.16.

Step 4 of our coding process—add the components to the container—is performed on lines 53–62. The `GridLayout` class's `add` method, presented in Table 11.5, is invoked on our container object `grid` to do this. The first argument passed to this method is the name of the component object to be added to a cell in the container. The other four arguments are the column and row number of the cell to place the component in, followed by the number of columns and rows to display the component in. These four arguments have been aligned in columns for instructional purposes.

Examining the progression of the value of the second argument as we move down through the invocations on lines 53–62, it is clear that the components have been added to the container in row number order. In addition, if we examine the progression of the value of the first argument for each of the rows, it is clear that the components have been added to the container's rows in column number order. This is consistent with the preferred coding style previously discussed.

```
1    import javafx.application.Application;
2    import javafx.event.*;
3    import javafx.geometry.*;
4    import javafx.scene.Scene;
5    import javafx.scene.control.*;
6    import javafx.scene.layout.*;
7    import javafx.scene.text.Font;
8    import javafx.stage.Stage;
9
10   public class AddingMachine extends Application
11   {
12      Label description, plus, equals, sum, a, b;
13      TextField aValue, bValue;
14      Button compute, clear;
15
16      @Override
17      public void start(Stage primaryStage)
18      {
19         //Step1: Declare the component objects, Step2: Set their properties
20         description = new Label("Computes a + b");
21         description.setFont(new Font("Arial", 24));
22         aValue = new TextField();
23         aValue.setPrefSize(120, 30);
24         plus = new Label("+");
25         plus.setFont(new Font("Arial", 24));
26         bValue = new TextField();
27         bValue.setPrefSize(120, 30);
28         equals = new Label("=");
29         equals.setFont(new Font("Arial", 24));
30         sum = new Label("x,xxx.xx");
31         sum.setFont(new Font("Arial", 24));
32         sum.setMinWidth(68);
33         a = new Label("a");
34         a.setFont(new Font("Arial", 24));
35         b = new Label("b");
36         b.setFont(new Font("Arial", 24));
37         compute = new Button(" Compute ");
38         clear = new Button ("  Clear   ");
39         Tooltip.install(clear, new Tooltip("Clears operands and result"));
40
```

```
      //Step 3: Declare the component container, and set its properties
      GridPane grid = new GridPane();
      grid.setHgap(10);
      grid.setVgap(10);
      grid.setPadding(new Insets(10, 0, 10, 10));
      grid.setHalignment(description, HPos.CENTER);
      grid.setHalignment(a, HPos.CENTER);
      grid.setHalignment(b, HPos.CENTER);
      grid.setHalignment(compute, HPos.CENTER);
      grid.setHalignment(clear, HPos.CENTER);

      //Step 4: Add the components to the component container
      grid.add(description, 0, 0, 5, 1); //col, row, nCols, nRows
      grid.add(aValue,      0, 1, 1, 1);
      grid.add(plus,        1, 1, 1, 1);
      grid.add(bValue,      2, 1, 1, 1);
      grid.add(equals,      3, 1, 1, 1);
      grid.add(sum,         4, 1, 1, 1);
      grid.add(a,           0, 2, 1, 1);
      grid.add(b,           2, 2, 1, 1);
      grid.add(compute,     0, 3, 1, 1);
      grid.add(clear,       2, 3, 1, 1);

      Scene scene = new Scene(grid); // add the container to the scene

      primaryStage.setTitle("Adding Machine");
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   public static void main(String[] args)
   {
      launch(args);
   }
}
```

Figure 11.17
The application `AddingMachine`.

11.5 EVENT PROCESSING

After the graphical interface is built, the next step in the GUI programming process is to identify the components in the interface that require application-dependent processing to be performed when the user interacts with them. For our adding machine, these would be the Compute and Clear buttons. When the Compute button is clicked, the two text field entries should be added, and the result output. A click of the Clear button should clear the text boxes and the output sum.

In GUI jargon, when the user interacts with a component on the interface, we say that an action has been performed on the interface, or that an event has occurred. A click on one of our buttons is an example of an event. Other examples include the completion of an entry into a text box denoted by the striking of the Enter key, the movement of the mouse pointer over the window, or simply a click into a text box.

GUI events are detected by the Java Runtime Environment and some of them are dealt with, or processed, with no effort on the programmer's part. For example, when the user of a program's GUI interface clicks into one of its text fields, the insertion point cursor (*caret*) appears in the text field. This event is processed, or handled, by the code of API methods associated with the text field. When the click event occurs, the Runtime Environment executes a process to *notify* these methods that the event has occurred. After being notified of the event, they execute and display the insertion caret in the text field. Other sections of code associated with the text field subsequently handle keystroke events by displaying a typed character in the text field, and moving the caret to the right.

Events such as a click into a text box can be handled by the API methods because the action to be taken (display the caret at the position of the click) when the event occurs is part of the predefined look and feel of the GUI component. These application *independent* GUI events are always processed by API methods. Other GUI events that occur on an application's interface, such as the clicking of the Compute button on our adding machine's interface, require the application programmer to write code to process the event because the action to be taken (perform the addition) is specific to the application.

More accurately, the application programmer partially processes these types of events because most often some processing that is part of the look and feel of the component also needs to be performed. For example, when the user clicks the Compute button of our adding machine, the API responds to the event by executing code that makes the button appear to have been depressed, because this is part of a predefined look and feel of a button. Then, the application performs the processing specific to it: fetch and parse the inputs, perform the addition, and display the result.

To initiate application-dependent processing when a GUI event occurs, the Java event-handling process permits the application programmer to add a method to the application that will be invoked when the event occurs. We say that the programmer can add to the list of methods that are *listening* for the event to occur, and these methods are *generically* referred to as *event handlers*. Figure 11.18 illustrates this concept and the event processing execution path it produces.

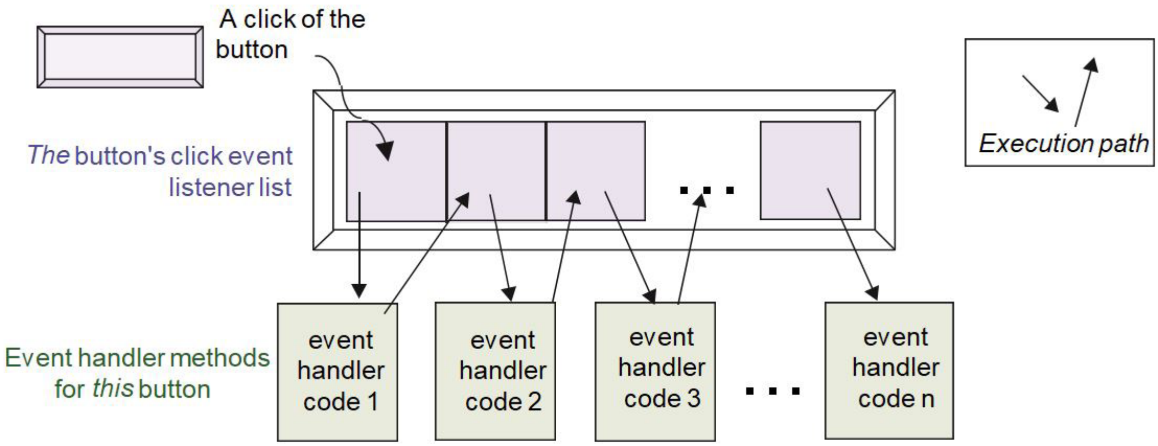


Figure 11.18
Java's event-processing process.

The process can be thought of as the execution of a switch statement, which executes the cases relevant to the event.

NOTE *Each component object added to the interface has its own event listener lists.*

The action of adding an application-dependent event handler method to a component's listener list is referred to as *registering* the event handler method. Below is a summary of the event processing terminology we have discussed.

Event: An action that occurs during the execution of a program. Examples include a click on a button object that is part of its scene, a mouse movement across the program's scene, or a keystroke when the keyboard's focus is the program's window.

Event handler: The code to be executed when event occurs.

Registering an event handler: The process of associating a particular event handler method, with a particular action on a particular object.

11.5.1 Implementing an Event Handler

The techniques to implement an event handler method and register it have evolved as new versions of Java were released. Currently there are six different coding techniques supported, and some of the syntax used within these techniques differs between Swing and FX GUI applications. The coding style we will use will cause a method coded within the application class whose name reflects the event that occurred, to be invoked when the event occurs. One example is a method named `computeClickHandler`. The application-dependent processing will be coded inside that method.

When an event handler method is invoked by the Java Runtime Environment, it is passed an object containing information about the event. This object is called the event object. Although most of the time the name of its class differs between FX and Swing based applications, in the case of a button event handler the object's class name is the same, `ActionEvent`. The signature of the method we will use to perform the application-dependent process when the Compute button in our adding machine application GUI (Figure 11.16) is clicked will be:

```
public void computeClickHandler(ActionEvent e)
```

The right column of Table 11.6 presents the event handler method parameter type for events on buttons, the keyboard keys, and the mouse. Its left column presents the more popular events on these three components.

As previously stated, the argument passed to the event handler contains information about the event. This information can be fetched by invoking methods in the parameter's class on it. In the case of a button, the `ActionEvent` class's `getSource` method returns the address of the `Button` object on which the event occurred. This method is used to determine which button was clicked when multiple buttons are registered with the same event handler method. We will show an example of its use later in this chapter.

Table 11.6
Event Handler Method Parameter Type for Events on Buttons, the Keyboard, and the Mouse

Type of Event (Action)	Event Handler's Parameter Type
On Button Objects The button is clicked	ActionEvent
On the Keyboard A keyboard key is typed (pressed and released), or a key is pressed, or a key is released	KeyEvent
On a Mouse The mouse is clicked, or the mouse is dragged, or the mouse is moved	MouseEvent

To complete the coding of the Compute button's event handler within our application `AddingMachine` shown in Figure 11.17, we have to fetch the two string inputs contained in the text boxes named `aValue` and `bValue`, parse them, perform the addition, and then output the result to the label named `result`. The code to do this within the event hander is show below.

```
public void computeClickHandler(ActionEvent e)
{
    String s;
    double a, b, result;
    DecimalFormat f = new DecimalFormat("#,##0.00");

    s = aValue.getText();
    a = Double.parseDouble(s);
    s = bValue.getText();
    b = Double.parseDouble(s);
    result = a + b;
    sum.setText(f.format(result));
}
```

11.5.2 Registering the Event Handler

To execute the above method when the Compute button is clicked, the method must be registered with the Java Runtime Environment as the Compute button's *click event* handler. To resister an event on any type of GUI component in an FX application, we invoke a method whose name begins with the characters `setON` on the component object. In our application the name of the button object is `compute`.

The characters in remainder of the method's name reflect the event that will be handled. As indicated in the left column of Table 11.6 the only *action* that can be performed on a button is a click, and so the suffix used is simply `Action` making the name of the method `setOnAction`.

Finally, `setOnAction`, like all event handler registration methods, contains one parameter. The argument passed to it is used to convey the class in which the Java FX event handler method named `handle` is coded, or the *actual code* of that method.

NOTE *When an event occurs within an FX application, the Java Runtime Environment always invokes the abstract method* `handle` *whose signature is defined in the interface* `EventHandler`.

To convey the *class* within the application that implements the method `handle`, the application passes an instance of that class to the method `setOnAction`. Then when the event occurs the runtime environment invokes the method on that object. The implication here is that the method coded in the application class to perform the application-dependent processing must be named `handle`, and the object passed to the `setOnAction` method must be an instance of the class in which this method is coded. The class can be the application's class or an inner class declared within it, and the class's heading must indicate that it implements the interface `EventHandler`.

Alternately, to convey the *code* of the event handler method `handle`, the application passes a *Lambda* expression to the `setOnAction` method that contains the event handler method's code. Normally the Lambda expression simply contains one line of code that is an invocation of a method coded in the application class that performs the application-dependent processing. Now the programmer is free to choose a name for that method that is representative of the event it services, e.g., `computeClickHandler`. As a result this technique is more commonly used to register the event handler, and we will use this alternative in the remainder of this chapter.

The code to register the application's Compute button's event handler method such that the method `computeClickHandler`, coded within the application, is invoked when the event occurs is:

```
compute.setOnAction( (e) -> computeClickHandler() );
```

The code `(e) -> computeClickHandler()` is the Lambda expression, and the Java statements to the right of the -> token in the Lambda expression become the code of the method `handle`. In this case there is only one statement, an invocation of the method `computeClickHandler`. We are now free to add this appropriately named method to our application, and place the code to service the event inside of it. We will gain a deeper understanding of Lambda expressions and their syntax in Sections 12.5 and 13.5.

Completing the Adding Machine Application

Figure 11.19 shows the application `AddingMachineV2`, which is the completed version of our adding machine. It is the program shown in Figure 11.17 with the previously discussed Compute button's event handler method, and the Clear button's handler method added to it (lines 73–85 and 87–93 respectively). The registering of these components' event handlers is performed on lines and 37 and 39 respectively. Because we have used Lambda expressions on these lines, the class's heading does not contain an implements clause.

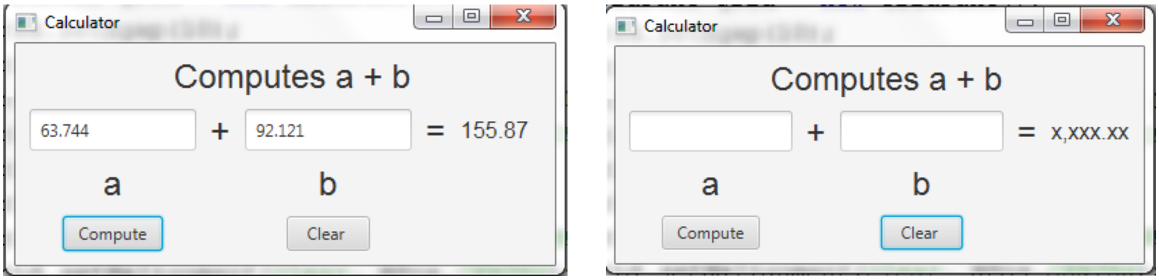
Figure 11.20(a) shows the application's GUI interface after the user entered two numbers to be added and then clicked the Compute button. Figure 11.20(b) shows the interface after a subsequent click on the Clear button.

```
1    import java.text.DecimalFormat;
2    import javafx.application.Application;
3    import javafx.event.*;
4    import javafx.geometry.*;
5    import javafx.scene.Scene;
6    import javafx.scene.control.*;
7    import javafx.scene.layout.*;
8    import javafx.scene.text.Font;
9    import javafx.stage.Stage;
10
11   public class AddingMachineV2 extends Application
12   {
13      Label description, plus, equals, sum, a, b;
14      TextField aValue, bValue;
15      Button compute, clear;
16
17      @Override
18      public void start(Stage primaryStage)
19      {
20         //Step1: Declare the component objects, Step2: Set their properties
21         description = new Label("Computes a + b");
22         description.setFont(new Font("Arial", 24));
23         aValue = new TextField();
24         aValue.setPrefSize(120, 30);        plus = new Label("+");
25         plus.setFont(new Font("Arial", 24));
26         bValue = new TextField();
27         bValue.setPrefSize(120, 30);        equals = new Label("=");
28         equals.setFont(new Font("Arial", 24));
29         sum = new Label("x,xxx.xx");
30         sum.setFont(new Font("Arial", 16));
31         sum.setMinWidth(68);
32         a = new Label("a");
33         a.setFont(new Font("Arial", 24));
34         b = new Label("b");
35         b.setFont(new Font("Arial", 24));
36         compute = new Button(" Compute ");
```

```
        compute.setOnAction( (e) -> computeClickHandler() );
        clear = new Button   ("  Clear   ");
        clear.setOnAction( (e) -> clearClickHandler() );
        Tooltip.install(clear, new Tooltip("Clears operands and result"));

        //Step 3: Declare the component container, and set its properties
        GridPane grid = new GridPane();
        grid.setHgap(10);
        grid.setVgap(10);
        grid.setPadding(new Insets(10, 0, 10, 10));
        grid.setHalignment(description, HPos.CENTER);
        grid.setHalignment(a, HPos.CENTER);
        grid.setHalignment(b, HPos.CENTER);
        grid.setHalignment(compute, HPos.CENTER);
        grid.setHalignment(clear, HPos.CENTER);

        //Step 4: Add the components to the component container
        grid.add(description, 0, 0, 5, 1);
        grid.add(aValue, 0, 1, 1, 1);
        grid.add(plus, 1, 1, 1, 1);
        grid.add(bValue, 2, 1, 1, 1);
        grid.add(equals, 3, 1, 1, 1);
        grid.add(sum, 4, 1, 1, 1);
        grid.add(a, 0, 2, 1, 1);
        grid.add(b, 2, 2, 1, 1);
        grid.add(compute, 0, 3, 1, 1);
        grid.add(clear, 2, 3, 1, 1);


       Scene scene = new Scene(grid); // add the container to the scene

        primaryStage.setTitle("Calculator");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    private void computeClickHandler()
    {
        String s;
        double a, b, result;
        DecimalFormat f = new DecimalFormat("#,##0.00");

        s = aValue.getText();
        a = Double.parseDouble(s);
        s = bValue.getText();
        b = Double.parseDouble(s);
        result = a + b;
        sum.setText(f.format(result));
    }
```

```

    private void clearClickHandler()
    {
       aValue.setText("");
       bValue.setText("");
       sum.setText("x,xxx.xx");
    }

    public static void main(String[] args)
    {
       launch(args);
    }
}
```

Figure 11.19
The application `AddingMachineV2`.



(a) (b)

Figure 11.20
The application `AddingMachineV2`'s window after a Compute button click followed by a Clear button click.

Alternate Event Processing Coding Style

In the application `AddingMachineV2` we coded two event handler methods, one for the Compute button and one for the Clear button, to perform the processing associated with a click event on those components. An alternate, and often used, coding style is to place the code bodies of all of the application's button handler methods inside one method The method would be named to imply that it performs the application-dependent processing for a click event on any of the buttons contained in the GUI interface, e.g., `buttonClickHandler`.

Inside this method, the code bodies of each button's event handler would be placed inside a different clause of an `if-else` statement. Each clause's Boolean condition would determine if the button whose event handler code body it contains was clicked.

As previously mentioned, the Java Runtime Environment invokes the method `handle` in response to an event on a GUI component. When it does, it passes that method an object that contains information about the event. In the case of a button click, that object is an instance of the class `ActionEvent`, and the information includes the address of the `Button` object that was clicked. To fetch that address, the `ActionEvent` class's inherited `getSource` method is invoked on the argument passed to the method `handle`.

The Java statements to the right of the -> token in our Lambda expression become the code of the method `handle`. To access the `ActionEvent` argument passed to `handle` by the Java Runtime Environment within that code, we use the variable name coded to the left of the -> token, which is named `e` in the Lambda expressions coded on lines 37 and 39 of Figure 11.19. To pass this argument on to our multi-button event handler method `buttonClickHandler` when `handle` invokes it, we recode the invocation in the Lambda expressions on lines 37 and 39 of Figure 11.19 to be:

```
(e) -> buttonClickHandler(e)
```

and the signature of our multi-button event handler would be:

```
private void buttonClickHandler(ActionEvent e)
```

Figure 11.21 shows the code of this method. It provides the functionality of the two event handlers on lines 73–93 of Figure 11.19. The signature of the method includes a parameter named `e` of type `ActionEvent` to receive the argument passed to the method from its invocation coded in the revised Lambda expression.

The `getSource` Method

The Boolean condition in the `if` statement on line 7 of Figure 11.21 invokes the `getSource` method on `ActionEvent` object `e` to determine if the Compute button was clicked. If that is the case, it executes the same code presented on lines 73–84 of Figure 11.19 to service the button click. Otherwise it executes the `else` clause's code block, which contains the same code presented on lines 90–92 of Figure 11.19, to service a click on the Clear button.

```
1   public void buttonClickHandler(ActionEvent e)
2   {
3      String s;
4      double a, b, result;
5      DecimalFormat f = new DecimalFormat("#,##0.00");
6
7      if(e.getSource() == compute)
8      {
9         s = aValue.getText();
10        a = Double.parseDouble(s);
11        s = bValue.getText();
12        b = Double.parseDouble(s);
13        result = a + b;
14        sum.setText(f.format(result));
15     }
15     else
17     {
18        aValue.setText("");
19        bValue.setText("");
20        sum.setText("x,xxx.xx");
21     }
22  }
```

Figure 11.21
The alternate coding style for the application `AddingMachineV2` event handlers.

11.5.3 Drawing 2-D Shapes

A paint event is a generic term for any event that causes a graphical object to be drawn or redrawn. The most obvious paint events are the initial display of an application's GUI window and maximizing a window after it has been minimized. A more subtle paint event is continuously redrawing a window and all of its GUI components as it is dragged across the monitor.

In addition to the GUI components we have studied so far, Java FX provides an assortment of 2-D shapes that can be displayed within a window. Like the other GUI components we have studied, they are drawn/redrawn when a paint event occurs.

There are two techniques available for adding 2-D shapes to a window. One of these uses the same four-step approach we have used to add control objects (e.g., buttons, labels, etc.) to a window: create the component objects, set their properties, declare a container, and add the objects to it. However, when the component objects are 2-D shapes the objects created in the first step are instances of a class that inherits from the class `Shape`. The following code draws a 50 unit radius orange `Circle` object whose center is (150, 125) units from the upper left corner of a light pink colored scene.

```
Pane root = new Pane();

Circle circle = new Circle(150, 125, 50); //xc, yc, radius
circle.setFill(Color.ORANGE);
root.getChildren().add(circle);

Scene scene = new Scene(root, 300, 250);
scene.setFill(Color.LIGHTPINK);
```

The second technique used to display 2-D shapes in a window employs an art studio analogy. Within this analogy a rectangular canvas is created, and then a brush is created and associated with the canvas. The brush is used to draw shapes on the canvas. The canvas is an instance of the class `Canvas`, and the brush is an instance of the class `GraphicsContext`. The brush is created and associated with the canvas by invoking the method `getGraphicsContext2D` on the canvas and assigning the returned address to the brush. The shapes are drawn on the canvas by invoking 2-D shape methods defined in the `GraphicsContext2D` class on the brush. To display the shapes in an application's window, the canvas must be added to the window's container.

The below code draws a circle filled with the default color black, using a brush named `brush`, on a canvas named `canvas`.

```
1  Canvas canvas = new Canvas(40, 90); //(width, height) of the canvas
2  canvas.setLayoutX(280);
3  canvas.setLayoutY(40);
4  GraphicsContext brush = canvas.getGraphicsContext2D();
5  brush.fillOval(0, 51, 37, 37); //xULcorner, yULcorner, w, h
```

Line 1 creates our canvas. It is declared to be an instance of the class `Canvas` 40 units wide and 90 units high named `canvas`. This is the object we will draw on. Lines 2 and 3 specify the x and y coordinates of the upper left corner of the canvas. When the canvas is added to a `Pane` container, this is where its upper left corner will be placed relative to the upper left corner of the container. Details of these two methods are given in Table 11.4.

Line 4 associates the `GraphicContext` variable `brush` with the canvas declared on line 1 by storing the address of the canvas' `GraphicsContext` in the variable `brush`. This object is the brush we will use to draw on the canvas.

Line 5 uses that brush to draw a circle (actually an oval whose width and height are both 37 units) on the canvas by invoking the `fillOval` method defined in the `GraphicsContext` class. The (x, y) location of upper left corner of the rectangle that encloses it, relative to the upper left corner of the canvas object, is specified to be (0, 51).

The circle is drawn is black because the default fill color property of a `GraphicsContext` is black. To change that property of our brush to a different color, we can invoke the `GraphicsContext` method `setFill` on it *before* a filled shape is drawn, and pass `setFill` a static constant defined in the `Color` class. To change our circle's fill color to white the invocation would be:

```
brush.setFill(Color.WHITE);
```

2-D shapes can also be drawn unfilled using a line (called a stroke) to outline their perimeter. To do this, the prefix `fill` in the name of the method used to draw the shape is changed to `stroke`: e.g., `strokeOval`. In addition, the suffix used in the method to change the brush's line drawing color is changed to `Stroke`: e.g., `setStroke`. Unlike a paintbrush used to draw shapes on a piece of paper, a `GraphicsContext` brush can be dipped into two different colors at the same time. One color is used when drawing filled shapes, and the other color is used when drawing outlined (stroke) shapes.

Table 11.7 presents commonly used methods defined in the `GraphicsContext` class to draw shapes, text, and images on a canvas and set their properties. The first four entries in the table have stroke counterparts.

Table 11.7
Commonly Used Methods in the `GraphicsContext` class

Method Signature **Drawing on the canvas**	**Description** *** Indicated that there is a corresponding Stroke method**
`fillRect(double x, double y, double width, double height)`	Draws a filled rectangle on the canvas whose upper left corner is located at (`x`, `y`), `width` wide and `height` high*
`fillPolygon(double[] x, double[] y, int numOfPoints)`	Draws a filled polygon on the canvas. `x` and `y` are parallel arrays that contain the (`x`, `y`) coordinates of the `numOfPoints` verticies*
`fillOval(double x, double y, double width, double height)`	Draws an filled oval `width` wide and `height` high on the canvas whose enclosing rectangle's upper left corner is located at (`x`, `y`)*

(Contd.)

Method Signature	Description
Drawing on the canvas	*** Indicated that there is a corresponding Stroke method**
`fillText(String theText, double x, double y)`	Draws the string `theText` with its upper left edge at (`x`, `y`) in the current font.*
`StrokeLine(double x1, double y1, double x2, double y2)`	Draws a line using the current stroke line width from (`x1`, `y1`) to (`x2`, `y2`)
`drawImage(Image img, double x, double y)`	Draws the image stored at the specified path `img` on the canvas at (`x`, `y`). Image types supported are BMP, JIF, JPEG, and PMP
Setting properties	
`setFill(Paint color)`	Sets the current fill color of a fill shape to `color`
`setStroke(Paint color)`	Sets the current line color of a stroke shape to `color`
`setLineWidth(double width)`	Sets the current line width of stroke shape to `width`
`setFont(Font theFont)`	Sets the current font to `theFont`

The draw method that begins on line 46 of the class `SnowmanOnACanvas` shown in Figure 11.22 draws a snowman on a 40 x 90 pixel `Canvas` object it creates on line 48. When an instance of this class is created its constructor, which begins on line 11, is passed an x and y coordinate and the color of the snowman's hat. These are stored in three of the object's data members on lines 13–15. Then line 16 of the constructor invokes the class's `draw` method, defined on lines 46–62 to draw the snowman on the canvas.

The `draw` method begins by declaring a 40 by 90 pixel `Canvas` object on line 48 and storing its address in the data member `canvas`. Then lines 49–50 set the initial (x, y) location where the canvas will be placed when it is added to a non-layout container, to the values stored in the data members x and y. Line 52 declares the `GraphicsContext` variable `brush` and associates it with the canvas object `canvas`. This will be the `draw` method's brush.

Lines 53–61 draw the snowman shown in Figure 11.23 on the canvas referenced by data member `canvas`, using the techniques and the methods in Table 11.7 that we have discussed. After setting the current fill color (line 53) and stroke color of our brush (lines 53–54), the snowman's body and head are drawn as filled circles (lines 55 and 57) outlined by slightly larger stroke circles (lines 56 and 58). Then the current fill and stroke colors are changed to the data member `hatColor` (line 59), and the hat and brim are drawn as filled rectangles (lines 60–61).

```
1    import javafx.scene.canvas.Canvas;
2    import javafx.scene.canvas.GraphicsContext;
3    import javafx.scene.paint.Color;
4
```

```
public class SnowmanOnACanvas
{
   private int x, y;
   private Color hatColor;
   private Canvas canvas;

  public SnowmanOnACanvas(int intialX, int intialY, Color hatColor)
   {
      this.x = intialX;
      this.y = intialY;
      this.hatColor = hatColor;
      draw();
   }

   public int getX()
   {
      return x;
   }

   public void setX(int newX)
   {
      x = newX;
      canvas.setLayoutX(x);
   }

   public int getY()
   {
      return y;
   }

   public void setY(int newY)
   {
      y = newY;
      canvas.setLayoutY(y);
   }

   public Canvas getCanvas()
   {
      return canvas;
   }

   public void draw()
   {
      canvas = new Canvas(40, 90); //Width and height of the canvas
      canvas.setLayoutX(x); //X position of the Canvas' upper left corner (UL)
      canvas.setLayoutY(y); //Y

      GraphicsContext brush = canvas.getGraphicsContext2D();
      brush.setFill(Color.WHITE);
```

```
        brush.setStroke(Color.BLACK);
        brush.fillOval(0, 51, 37, 37); //Body's xUL, yUL, w, h
        brush.strokeOval(0, 49, 38, 37);
        brush.fillOval(9, 32, 18, 18); //Head
        brush.strokeOval(9, 32, 21, 19);
        brush.setFill(hatColor);
        brush.fillRect(10, 28, 20, 6); //Hat brim
        brush.fillRect(14, 17, 12, 15); //hat
    }
}
```

Figure 11.22
The class `SnowmanOnACanvas`.

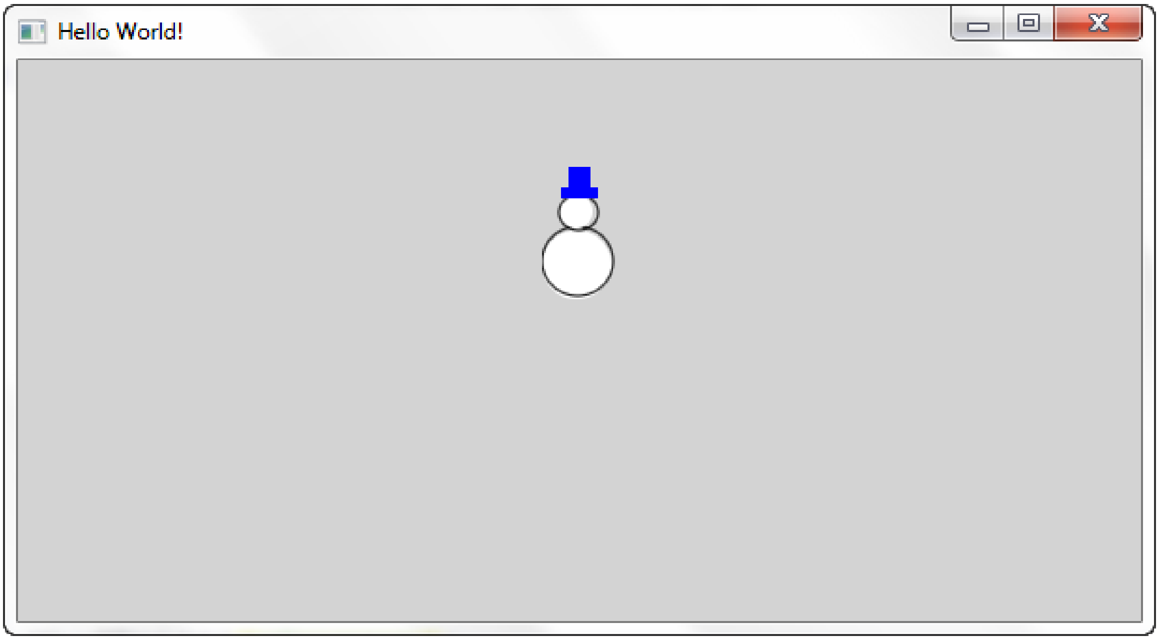


Figure 11.23
The window displayed by the application `DrawingShapes1`.

The application `DrawingShapes1` shown in Figure 11.24 displays the window shown in Figure 11.23. Line 15 declares an instance of the class `SnowmanOnACanvas` passing its constructor the canvas's initial (x, y) location (280, 40), and the snowman's hat color: blue.

Pane container `root` is created on line 18. The code `sm1.getCanvas()`on the right side of line 21 returns the address of the canvas with the drawing on it, and then adds that component's address to the container `root`. Then the container `root` is added to the scene (line 23), and the scene is added to the stage (line 26), and the stage is displayed (line 27). Since a Pane is a non-layout container, the upper left corner of the canvas is positioned at its specified (x, y) location relative to the pane's upper left corner.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.*;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class DrawingShapes1
{
   private SnowManOnACanvas sm1;

   public void start(Stage primaryStage)
   {
      //Step1: declare the components, Step2: set their properties
      sm1 = new SnowManOnACanvas(280, 40, Color.BLUE);

      // Step 3: Declare the container and set its properties
      Pane root = new Pane();

      // Step 4: Add the components to the component container
      root.getChildren().addAll(sm1.getCanvas());

      Scene scene = new Scene(root, 600, 300, Color.LIGHTGRAY);

      primaryStage.setTitle("Drawing On a Canvas!");
      primaryStage.setScene(scene);
      primaryStage.show();
   }
}
```

Figure 11.24
The application `DrawingShapes1`.

11.5.4 Keyboard, Mouse, and Key Frame Timer Events

Many of the GUI components included in the API respond to mouse-click events and keyboard events such as the pressing or typing of a key. The most obvious examples of this are radio-button event handlers and check-box event handlers that change the appearance of these components when they are clicked. Another example is the appearance of the caret in a text field when it is clicked, and the subsequent display of typed characters in the text field to the left of the caret. The GUI support in the API includes implementations of these non-application-dependent event handlers and their registrations.

Just as GUI components respond to these events to maintain their look and feel, there is often the need for an application to respond to a mouse or keyboard event in an application-dependent manner. Application-dependent processing of a `Button` click event has already been discussed in this chapter. Other mouse events, such as the dragging of the mouse or clicking on a portion of the application's window that does not contain a GUI component may be an important event in a particular application.

For example, a mouse-drag event could be used to move the snowman shown in Figure 11.23 to another location in the program's window, or a key-typed event such as typing the cursor control (arrow) keys could be used to move the snowman around the window. Similarly, a timer-tick event is certainly an important event to many applications that are time dependent.

In this section, we will learn how to incorporate keyboard, timer, and mouse events into any Java application. Although we have reacted to some of these events in our game programs, the details of the keyboard, timer, and mouse event handler implementations and their registration was performed by the game environment. Essentially, in the remainder of this section, we will gain insights into how this was accomplished by the game environment, so we can process these events in any application we write.

Keyboard and mouse events are processed within an application using the techniques discussed in Sections 11.5.1 and 11.5.2. As mentioned there, when an event occurs within an FX application, the runtime environment invokes a one-parameter method named `handle` to process events and we must overwrite that method. The technique we have used thus far to do this is to invoke a method whose name begins with `setOn` on the component the event occurred on, and pass it a Lambda expression. The Lambda expression becomes the code of the method `handle`.

The prefix `setOn` is followed by a suffix that is dependent on the event to be serviced. As discussed in Section 11.5.2, when the event is a button click, the suffix is `Action`, and the name of the method is therefore `setOnAction`. Table 11.8 presents the names of the analogous methods to invoke for keyboard events and two, of the eleven possible, mouse events. These methods are invoked on the application's `Scene` object. The table also includes the method `setOnAction` for comparative purposes.

Table 11.8
A Subset of FX Event Handler Registration Methods

Event, and the Event Object Generated	Method
A key is *pressed*, generates a KeyEvent	`setOnKeyPressed(aLambdaExpression)`
A key is *released*, generates a KeyEvent	`setOnKeyReleased(aLambdaExpression)`
A key is *typed* (pressed & released), generates a KeyEvent	`setOnKeytyped(aLambdaExpression)`
The mouse is clicked, generates a MouseEvent	`setOnMouseClicked(aLambdaExpression)`
The mouse is dragged, generates a MouseEvent	`setOnMouseDragged(aLambdaExpression)`
A button is clicked, generates an ActionEvent	`setOnAction(aLambdaExpression)`

Keyboard Events

A key-pressed event occurs when a keyboard key is struck, and a key-released event occurs when a key is released. In addition, a key-typed event occurs only when a *non-action* key is struck. The action keys include the keys Shift, Num Lock, End, Home, Caps Lock, function keys, arrow keys, Page-up, Page-down, Backspace, etc.

Whenever any key is held down, it repeatedly generates a key-pressed event. If the key is a non-action key, it also continually generates a key-typed event. In either case, when the key is released, a single key-released event occurs.

When the event handler services a key-typed event the `getCharacter()`method, invoked on the `KeyEvent` argument passed to the event handler method, returns a string containing the character (lower or upper case) generated by the non-action keys. The returned character can be used to determine which non-action key has been struck. The following key event handler performs its output when the key P (lower or upper case) is typed:

```
public void keyTypedHandler(KeyEvent e)
{
  if(e.getCharacter.equals("P") || e. getCharacter.equals("p"))
  {
    System.out.println("The P key was struck");
  }
}
```

When the event handler services a key-pressed or a key-released event the `getCode` method, invoked on the `KeyEvent` argument passed to the event handler method, returns a constant defined in the class `KeyCode` that is associated with the pressed or released key. The following event handler performs its output when the right arrow key is struck.

```
public void keyPressedHandler(KeyEvent e)
{
  if(e.getCode()  == KeyCode.RIGHT)
  {
    System.out.println("The right arrow key was struck");
  }
}
```

The application `Animation1` shown in Figure 11.25 is the application `DrawingShapes1` shown Figure 11.24, with a keyboard handler added to it. When the left or right arrow key is pressed the snowman, shown in Figure 11.23, is moved 4 units in the direction of the key pressed. Holding the key down continuously generates key-pressed events, causing the snowman to move smoothly across the window.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.*;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
```

```
import javafx.stage.Stage;

public class Animation1
{
   private SnowManOnACanvas sm1;

   public void start(Stage primaryStage)
   {

      //Step1: declare the components, Step2: set their properties
      sm1 = new SnowManOnACanvas(280, 40, Color.BLUE);

      // Step 3: Declare the container and set its properties
      Pane root = new Pane();

      // Step 4: Add the components to the component container
      root.getChildren().addAll(sm1.getCanvas());

      Scene scene = new Scene(root, 600, 300, Color.LIGHTGRAY);
      scene.setOnKeyPressed((e) -> keyEventHandler(e)); //any key

      primaryStage.setTitle("Annimation Using the Keyboard");
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   public void keyEventHandler(KeyEvent e)
   {
      int speed = 4;

      if(e.getCode() == KeyCode.RIGHT)
      {
         sm1.getCanvas().setTranslateX(sm1.getCanvas().
                                       getTranslateX() + speed);
      }
      else if(e.getCode() == KeyCode.LEFT)
      {
         sm1.getCanvas().setTranslateX(sm1.getCanvas().
                                       getTranslateX() - speed);
      }
   }

   public static void main(String[] args)
     {
      launch(args);
   }
}
```

Figure 11.25
The application `Animation1`.

Line 25 of the application invokes the Scene class's `setOnKeyPressed` method on the object `scene` and passes it a Lambda expression that becomes the code of the method `handle` that the runtime environment will invoke when a key-pressed event occurs. The code contains one statement, an invocation of the method `keyEventHandler` and passes it the `KeyEvent` object passed to the Lambda expression's parameter `e` when a key-pressed event occurs. The method `keyEventHandler` is coded on lines 32–44. The Boolean conditions of the if-else statement that begins on line 36 invoke the `getCode` method on the object `e` passed to the method to determine if the left or right arrow key was pressed.

Inside the if-else statement, lines 38 and 42 use the `getTraslateX` and `setTranslateX` methods to change the x location of the canvas the snowman was drawn on by `speed` (4) units. This causes the window to be redrawn with the canvas at a new, *current*, x location. Its *current* location is its *initial* x location (280 on line 16), set by the `setLayout` method (line 49 of Figure 11.22) before the canvas is added to the container on line 22 of Figure 11.25, plus the net amount it has been translated from that location.

To change the net amount it has been translated, lines 38 and 42 of Figure 11.25 invoke the `getCanvas` method on the `SnowManOnACanvas` object `sm1` to fetch the address of the `Canvas` object the snowman was drawn on. Then the `Canvas` class's inherited method `setTranslateX` is invoked on that object to change the net amount the canvas has been translated from its initial location. The `setTranslateX` method is passed a new value of the net x translation. The new value is the current value, fetched by the invocation of the `getTranslateX` method, plus or minus 4 units stored in the variable `speed` defined on line 34. Changing the translation of the canvas caused the window to be redrawn, with the canvas containing the snowman in its new position.

Mouse Events

As mentioned previously in this chapter, there are eleven events that can occur on a mouse. Two of the most common are the mouse-click and mouse-drag events. A mouse-click event occurs when either of the mouse buttons is released after it is pressed. A drag event occurs when either of the mouse buttons is pressed and then the mouse is moved.

To determine which button has been clicked or pressed, the `MouseEvent` class's `getButton` method can be invoked on the object passed to the event handler method. The method returns a constant defined in the `MouseButton` class that can be used to determine if the left button (`MouseButton.PRIMARY`) or right button (`MouseButton.SECONDARY`) caused the event. The following code segment can be used inside a mouse-click or mouse-dragged event handler to perform the processing associated with a left or right button click or drag action.

```
public void mouseButtonHandler(MouseEvent e)
{
   if(e.getButton() == MouseButton.PRIMARY)
   {
      //Code associated with the left button
   }
      else if(e.getButton() == MouseButton.SECODARY)
```

```
        {
           //Code associated with the right button
        }
     }
```

The application `Animation2` shown in Figure 11.26 is the application `DrawingShapes1` shown in Figure 11.24 with a mouse-clicked, and a mouse-dragged, event handler added to it. When the program is launched, it displays the window shown in Figure 11.23. After selecting the canvas on which the snowman is drawn by left clicking it, it can be dragged with the left mouse button to any location in the window. To end the drag, a left click is performed on any portion of the window outside the canvas.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.MouseButton;

public class Animation2 extends Application
{
   private SnowManCanvas sm1;
   double deltaX, deltaY;
   boolean sm1Clicked = false;

   public void start(Stage primaryStage)
   {

      sm1 = new SnowManCanvas(280, 40, Color.BLUE);

      Pane root = new Pane();

      root.getChildren().addAll(sm1.getCanvas());

      Scene scene = new Scene(root, 600, 300, Color.LIGHTGRAY);
      scene.setOnMouseClicked((e) -> mouseClickedHandler(e));
      scene.setOnMouseDragged((e) -> mouseDraggedHandler(e));

      primaryStage.setTitle("Drawing On a Canvas");
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   public void mouseClickedHandler(MouseEvent e)
   {
      if((e.getButton() == MouseButton.PRIMARY)) //a left click
      {
         if(e.getX() > sm1.getX() + sm1.getCanvas().getTranslateX() &&
```

```
38                e.getX() < sm1.getX() + sm1.getCanvas().getTranslateX() + 40 &&
39                e.getY() > sm1.getY() + sm1.getCanvas().getTranslateY() &&
40                e.getY() < sm1.getY() + sm1.getCanvas().getTranslateY() + 90)
41             { //the click was on the canvas
42                deltaX =  e.getX() - sm1.getCanvas().getTranslateX();
43                deltaY =  e.getY() - sm1.getCanvas().getTranslateY();
44                sm1Clicked = true;
45             }
46             else //the click was not on the canvas
47             {
48                sm1Clicked = false;
49             }
50          }
51       }
52
53       public void mouseDraggedHandler(MouseEvent e)
54       {
55         if(sm1Clicked == true && e.getButton() == MouseButton.PRIMARY)
56          {
57            sm1.getCanvas().setTranslateX(e.getX() - deltaX);
58            sm1.getCanvas().setTranslateY(e.getY() - deltaY);
59          }
60       }
61
62       public static void main(String[] args)
63       {
64          launch(args);
65       }
66    }
```

Figure 11.26
The application `Animation2`.

Lines 25 and 26 of the application invoke the Scene class's `setOnMouseClicked` and `setOn-MouseDragged` method respectively on the object scene, and passes them a Lambda expression. These Lambda expressions become the code of the method `handle` that the runtime environment will invoke when a mouse-click (line 25) or a mouse-drag (line 26) event occurs. Each Lambda expression contains one statement that invokes a method, coded within the application on lines 33–51 and 53–60, and passes these methods the `MouseEvent` object passed to them when their event occurs.

The method `mouseClickedHandler` (line 33) executes when a mouse-clicked event occurs. The Boolean condition of the `if` statement that begins on line 35 invokes the `getButton` method on the object `e` passed to the method to determine if the left button was clicked. If it was not, the Boolean class level variable `sm1Clicked` (that was declared on line 13) is set to `false`. If the left mouse button was clicked, the if-else statement that begins on line 37 executes. Lines 37–40 determine if the click occurred when the tip of the mouse pointer was within the perimeter of the canvas. If it was, the class level variable `sm1Clicked` is set to `true` on line 44. If it was not, `sm1Clicked` is set to `false` on line 48. This variable is used in the method `mouseDraggedHandler` on line 55 to determine if the canvas was selected to be dragged by a left button click.

Within the `if` statement's Boolean condition on lines 37–40, the `getX` and `getY` methods are invoked on the argument passed to the parameter `e` to fetch the (x, y) location of the mouse pointer's tip depicted as a triangle in Figure 11.27. The remainder of the code to the right of the Boolean relational operators on each of those four lines determines the current location of the edges of the canvas. Specifically, the remainder of the code on lines 37 and 38 determines the current values of the canvas's left and right edge x coordinates, and the remainder of the code on lines 39 and 40 determines the current values of the canvas's upper and lower edge y coordinates.

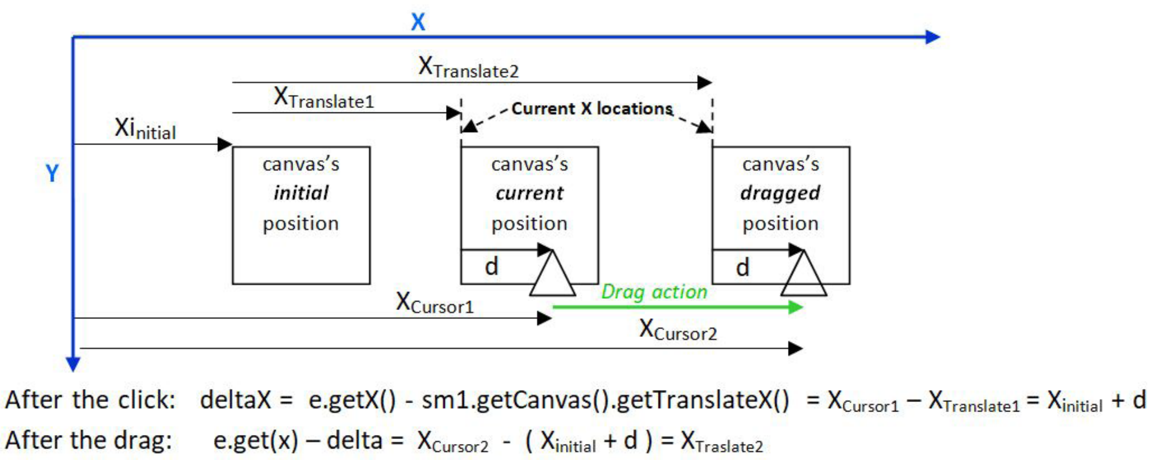


Figure 11.27
The canvas at its initial, current, and dragged x position.

As shown in Figure 11.27, a component's (the canvas's) *current* x location is equal to its *initial* x location plus the net amount it has been *translated* in the x direction. Lines 37–38 invoke the `SnowManCanvas`'s class's `getX` method on `sm1` to determine the x coordinate of the canvas's initial location, and they invoke the `Canvas`'s class's `getTranslateX` methods to determine the net amount the canvas has been translated in the x direction. A component's current y location is determined in a similar way.

Thus the below code within lines 37–40 determine the current (x, y) location of the upper left corner of the canvas, which is also the x location of its left edge and the y location of its upper edge.

```
sm1.getX() + sm1.getCanvas().getTranslateX()
sm1.getY() + sm1.getCanvas().getTranslateY()
```

Line 38 of Figure 11.26 adds 40, the width of the canvas, to the above first line, to determine the x coordinate of the canvas's right edge. Line 40 adds 90, the height of the canvas, to the above second line to determine the y coordinate of the canvas's lower edge. Thus the Boolean expression of line 37 determines if the curser position is to the right of the left edge of the canvas, and the

Boolean condition on line 38 determines if the cursor is to the left or the canvas's right edge. In a similar way, lines 39 and 40 determine if the cursor is between the canvas's upper and lower edge.

When this is the case, line 42 of Figure 11.26 subtracts the canvas's current translation ($X_{Translate1}$ in Figure 11.27) from the cursor's current x position ($X_{Cursor1}$ in Figure 11.27), and stores that value in the variable class level variable `deltaX`. As can be deduced by examining Figure 11.27 and the first equation at the bottom of the figure, this value is equal to the sum of the canvas's *initial* x position, $X_{initial}$, and the horizontal distance between the canvas's left edge and the tip of the mouse pointer (d) when the click occurred. Similarly, line 43 stores the sum of the canvas's *initial* y position and the vertical distance between the canvas's top edge and the tip of the mouse pointer when the click occurred in the variable class level variable `deltaY`.

Whenever the mouse is dragged one unit in any direction, the method `MouseDraggedHandler` that begins on line 53 executes. The first term of the Boolean condition of the `if` statement on line 55 verifies that the canvas was left clicked before the drag occurred. The second term verifies that the left mouse button is being used for the drag.

When both of these conditions are true, line 57 gets the sm1's Canvas object and invokes the `setTranslateX` method on it. This method changes the value of the canvas's x translation to the value passed to it, `e.getX() - deltaX`. Since `e.getX()` after the drag returns the new x coordinate of the mouse pointer referred to as $X_{Cursor2}$ in Figure 11.27, and `deltaX` is equal to $X_{Initial} + d$, the new value of the canvas's x translation is set to $X_{Cursor2} - (X_{Initial} + d)$.

Examining Figure 11.27, this can be deduced to be $X_{Translate2}$, the horizontal distance from the initial position of the canvas's left edge to the dragged position of the canvas's left edge. Similarly line 58 computes the canvas's new vertical translation. When the `MouseDraggedHandler` method ends, the application's pane is redrawn by the runtime environment using the new values of the canvas's x and y translation to position the canvas.

Key Frame Timer Events

A key frame timer event is an `ActionEvent` that occurs when a `KeyFrame` timer object's time interval expires. As is the case with the keyboard and mouse events we have discussed, when the event occurs the Java Runtime Environment invokes the method `handle` whose signature is defined in the interface `EventHandler`.

When a `KeyFrame` object is created, its time interval must be specified by passing an object in the class `Duration` to the first parameter of its constructor. In addition, a Lambda expression that becomes the code of the method `handle` can also be passed the constructor. The following line of code constructs a `KeyFrame` object whose time interval duration is 0.1 seconds, and whose event handler will invoke the method `kF1KeyFrameHandler`.

```
KeyFrame kF1 = new KeyFrame(Duration.seconds(0.1), e -> kF1KeyFrameHandler(e));
```

Its first argument is an invocation of the `Duration` class's static method `seconds` that is passed the duration of the key frame's time interval in seconds and returns a `Duration` instance that represents that time interval. The description of this method, and other commonly invoked methods within this argument, are shown in the top portion of Table 11.9. The lower portion of the table contains methods commonly used in the `Timeline` class.

A `KeyFrame` object is associated with a `Timeline` object by passing it to the `Timeline` object's constructor when it is created. Multiple key frames can be associated with one timeline, by passing them to the constructor. When one key frame is associated with a timeline, the timeline is a linear sequence of that key frame's timer events separated by the key frame's specified duration, as shown in Figure 11.28a. Each interval in the sequence is referred to as a cycle.

Table 11.9
Commonly Used Methods in the `Duration` and `Timeline` Classes

Methods Commonly Used Methods in the `Duration` Class	Description
`public Duration millis(double ms)`	Returns a `Duration` object that represents `ms` milliseconds
`public Duration seconds(double s)`	Returns a `Duration` object that represents `s` seconds
`public Duration minutes(double m)`	Returns a `Duration` object that represents `m` minutes
`public Duration hours(double h)`	Returns a `Duration` object that represents `h` hours
Commonly Used Methods in the Timeline Class	
`Timeline(KeyFrame keyFrame1, . . .)`	Constructs a `Timeline` object compose of the `KeyFrame` objects passed to it
`public void setCycleCount(int nCycles)`	Sets the number of cycles to be executed, before the `Timeline` object that invokes it is over, to `nCycles`
`public void play()`	Executes the `Timeline` object that invokes it beginning at the timeline's *current* position
`public void pause()`	Pauses the `Timeline` object that invokes it at the timeline's *current* position
`public void playFromStart()`	Executes the `Timeline` object that invokes it beginning at the timeline's initial position

The number of cycles of the key frames in a timeline is specified by invoking the `setCycleCount` method on the `Timeline` instance and passing it the number of cycles. The default is one cycle. When a cycle ends, the code in the Lambda expression of its key frame executes and then the next cycle begins. The cycle sequence is initialed by invoking `play` method on the timeline.

The below code creates the timeline tL1 shown in Figure 11.28a, consisting of two one-second duration cycles, and plays it. At the end of each cycle the method `kF1KeyFrameHandler`, is invoked.

```
KeyFrame kF1 = new KeyFrame(Duration.seconds(1.0), e ->
                                              kF1KeyFrameHandler(e));
Timeline tL1 = new Timeline(kF1);
tL1.setCycleCount(2);
tL1.play();
```

When multiple key frames are associated with a timeline, the timeline is the *parallel* execution of all of the key frames passed to the time line's constructor, as shown in Figure 11.28b. All of the cycles of all of these key frames begin at the same time, which dictates that all subsequent cycles of all of key frames begin when the cycle of the key frame with the longest duration time ends as shown in the figure.

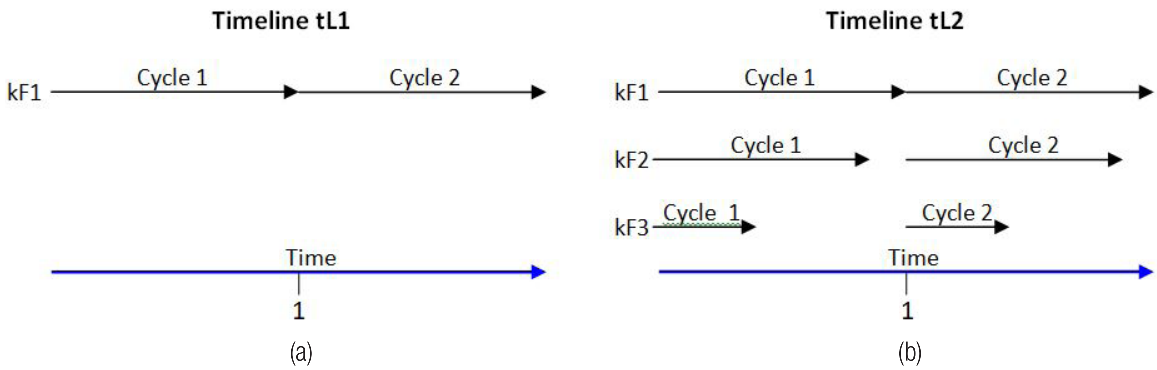


Figure 11.28
Times lines tL1 and tL2.

The below code creates the timeline tL2 shown in Figure 11.28b. It contains three key frames, kF1, kF2, and kF3. kF1's duration is 1 second, kF2's duration is 0.8 seconds, and kF3's duration is 0.5 seconds. When each cycle ends, three different methods are invoked.

```
KeyFrame kF1 = new KeyFrame(Duration.seconds(1.0),
                            e -> kf1KeyFrameHandler(e));
KeyFrame kF2 = new KeyFrame(Duration.seconds(0.8),
                            e -> kf2KeyFrameHandler(e));
KeyFrame kF3 = new KeyFrame(Duration.seconds(0.5),
                            e -> kf3KeyFrameHandler(e));
TimeLine tL1 = new TimeLine(kF1, kF2, kF3);
tL1.setCycleCount(2);
tL1.play();
```

Figure 11.29 presents the application `Animation3`. It is the application `DrawingShapes1` shown Figure 11.24 with a timeline added to it that animates the snowman's canvas. When the program is launched, it displays the window shown in Figure 11.30a, and then the snowman's canvas is smoothly translated diagonally to the position shown in Figure 11.30b. When it reaches the edge of the window, it is translated back to its original position. This is process is repeated until the cycle count, set on line 25, expires.

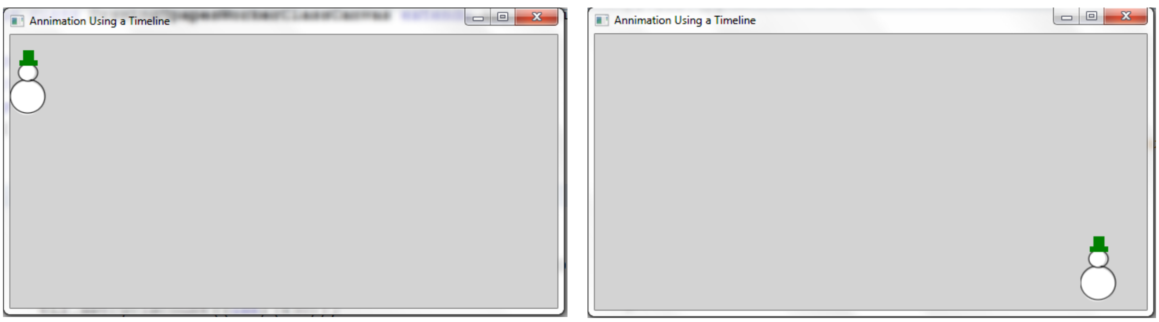
Line 18 of the application shown in Figure 11.29 declares an instance of the class `SnowManOn-Canvas`, with a green color hat positioned at (0, 0). Line 28 of the application creates a `Pane` object, and line 30 adds the snowman's `Canvas` object to that container.

```
1   import javafx.application.Application;
2   import javafx.scene.Scene;
3   import javafx.scene.layout.Pane;
4   import javafx.scene.paint.Color;
5   import javafx.stage.Stage;
6   import javafx.event.ActionEvent;
7   import javafx.util.Duration;
8   import javafx.animation.KeyFrame;
9   import javafx.animation.Timeline;
10
11  public class Animation3 extends Application
12  {
13     private SnowManCanvas sm1;
14     private double speedX, speedY;
15
16     public void start(Stage primaryStage)
17     {
18        sm1 = new SnowManCanvas(0, 0, Color.GREEN);
19        speedX = 4.0;
20        speedY = speedX/2.63;
21
22        KeyFrame kF1 = new KeyFrame(Duration.seconds(0.03),
23                                    e -> kF1KeyFrameHandler());
24        Timeline tL1 = new Timeline(kF1);
25        tL1.setCycleCount(430);
26        tL1.play();
27
28        Pane root = new Pane();
29
30        root.getChildren().addAll(sm1.getCanvas());
31
32        Scene scene = new Scene(root, 600, 300, Color.LIGHTGRAY);
33
34        primaryStage.setTitle("Animation Using a Timeline");
35        primaryStage.setScene(scene);
36        primaryStage.show();
37    }
39
40    public void kF1KeyFrameHandler(ActionEvent e)
41    {
42      if((sm1.getCanvas().getTranslateY() >= 215.0) ||
43         (sm1.getCanvas().getTranslateY() < -1))
```

```
44        {
45           speedX = -speedX;
46           speedY = -speedY;
47        }
48        sm1.getCanvas().setTranslateX(sm1.getCanvas().getTranslateX() +
                                        speedX);
49        sm1.getCanvas().setTranslateY(sm1.getCanvas().getTranslateY() +
                                        speedY);
50     }
51
52     public static void main(String[] args)
53     {
54        launch(args);
55     }
56  }
```

Figure 11.29
The application `Animation3`.

The `KeyFrame` object `kF1` is declared on lines 22–23. The constructor is passed a cycle duration of 0.03 seconds, and the Lambda expression that will become the code of the method `handle` invoked by the run time environment when an `ActionEvent` occurs. An `ActionEvent` occurs at the end of every cycle, in this case every 0.03 seconds. Lines 24 declares the `Timeline` object `tL1` and passes its constructor the `KeyFrame` object `kF1`. Lines 25–26 set the timeline's number of cycles to 430, and then plays the timeline. This starts the animation that continues for 13.9 seconds: 430 cycles x 0.03 seconds per cycle.



(a) (b)

Figure 11.30
The extremes of the canvas translation produced by the application `Animation3`.

The method invoked by the event handler `handle`, begins on line 40. The Boolean condition of the `if` statement that begins on line 42 determines if the left edge or bottom edge of the canvas is outside the scene. When it is, lines 45–46 reverse the sign of the values added to the canvas's current translation, `speedX` and `speedY`, which reverses the canvas's traversal direction. These two variables are initialized on lines 19 and 20.

11.6 THE PANE CONTAINER AND LAYOUT MANAGERS: A SECOND LOOK

The application `AddingMachine` presented in Figure 11.17 declared a `GridPane` layout manager to position GUI components in the scene. In this section we will discuss a popular alternative to using a layout manager to position a GUI's components. Then we will discuss four other often-used layout managers.

11.6.1 Working with the Pane Container

An instance of the class `Pane` is a container that GUI components can be added to, but it does not manage the positioning of the components added to it. Rather, the programmer specifies the (x, y) locations of the upper left corner of the components relative to the upper left corner of the `Pane` object before they are added to the `Pane` container. We had a glimpse of this approach within the animation applications previously discussed in this chapter.

The application `AddingMachineV3,` shown in Figure 11.31, uses this technique to recode the building of the GUI window, displayed by the application `AddingMachineV2,` presented in Figure 11.20. A component's (x, y) location is stored in two of its properties, and so it makes sense to assign their values in the portion of the application where the component properties are specified. In Figure 11.31, that is done on lines 32–73.

To specify the x location of a component's upper left corner relative to upper left corner of the container it will be added to, the `setLayoutX` method is invoked on the component and the value of the coordinate is passed to the method. Similarly the `setLayoutY` method is used to specify a component's y location. For example, lines 32 and 33 set the location of the label `description` declared on line 20.

When the program is run for the first time and its window is displayed, most often some of the (x, y) locations of the components need to be adjusted. After the adjustments are made, the application is run again and some additional adjustments are often necessary. To reduce the number of iterations in this process, the design of the interface, shown in Figure 11.15, should be drawn on a piece of graph paper. Then the (x, y) coordinates of the upper left corner of each component can be read from the graph paper and coded into the program.

Using this approach requires two more lines of code per component to specify its location, and a few iterations to tune up the locations. However, even when a layout manager is used a few iterations are usually required to adjust component spacing and centering within the container.

There is one other difference between this version of our adding machine and the earlier version. In this application all of the components are added to the container using the statement on lines 78 and 79. The `addAll` method, instead of the `add` method, is invoked on the pane container and the names of all of the components are passed to it as arguments. This is a preferred coding style, because it reduces the number of lines in the program. The GUI window it produces is shown in Figure 11.32

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.scene.control.*;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import java.text.DecimalFormat;

public class AddingMachineV3 extends Application
{
   Label description, plus, equals, sum, a, b;
   TextField aValue, bValue;
   Button compute, clear;

   @Override
   public void start(Stage primaryStage)
   {
      // Step 1, declare the component objects
      description = new Label("Computes a + b");
      aValue = new TextField();
      plus = new Label("+");
      bValue = new TextField();
      equals = new Label("=");
      sum = new Label("x,xxx.xx");
      a = new Label("a");
      b = new Label("b");
      compute = new Button(" Compute ");
      clear = new Button  ("  Clear   ");

      // Step 2, specify the components properties
      description.setLayoutX(120);
      description.setLayoutY(0);
      description.setPrefSize(300, 30);
      description.setFont(new Font("Arial", 24));
      aValue.setLayoutX(60);
      aValue.setLayoutY(50);
      aValue.setPrefSize(120, 30);
      aValue.setFont(new Font("Arial", 18));
      plus.setLayoutX(195);
      plus.setLayoutY(50);
      plus.setPrefSize(20, 30);
      plus.setFont(new Font("Arial", 24));
      equals.setLayoutX(365);
      equals.setLayoutY(50);
      equals.setPrefSize(20, 30);
      equals.setFont(new Font("Arial", 24));
      bValue.setLayoutX(230);
      bValue.setLayoutY(50);
```

```
        bValue.setPrefSize(120, 30);
        bValue.setFont(new Font("Arial", 18));
        sum.setLayoutX(395);
        sum.setLayoutY(50);
        sum.setPrefSize(120, 30);
        sum.setFont(new Font("Arial", 18));
        a.setLayoutX(105);
        a.setLayoutY(85);
        a.setPrefSize(20, 30);
        a.setPrefHeight(30);
        a.setFont(new Font("Arial", 24));
        b.setLayoutX(275);
        b.setLayoutY(85);
        b.setPrefSize(20, 30);
        b.setPrefHeight(30);
        b.setFont(new Font("Arial", 24));
        compute.setLayoutX(70);
        compute.setLayoutY(120);
        compute.setPrefSize(90, 25);
        compute.setOnAction(e -> computeClickHandler(e));
        clear.setLayoutX(242);
        clear.setLayoutY(120);
        clear.setPrefSize(90, 25);
        clear.setOnAction(e -> clearClickHandler(e));

        // Step 4, add the components to the window
        Pane pane = new Pane();
        pane.getChildren().addAll(description, aValue, plus, bValue, equals,
                                   sum, a, compute, clear);

        Scene scene = new Scene(pane, 500, 200);

        primaryStage.setTitle("Calculator");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void computeClickHandler(ActionEvent e)
    {
        String s;
        double a, b, result;
        DecimalFormat f = new DecimalFormat("#,##0.00");

        s = aValue.getText();
        a = Double.parseDouble(s);
        s = bValue.getText();
        b = Double.parseDouble(s);
        result = a + b;
```

```
         sum.setText(f.format(result));
      }

      public void clearClickHandler(ActionEvent e)
      {
         aValue.setText("");
         bValue.setText("");
         sum.setText("x,xxx.xx");
      }
      public static void main(String[] args)
      {
         launch(args);
      }
   }
```

Figure 11.31
The application `AddingMachineV3`.

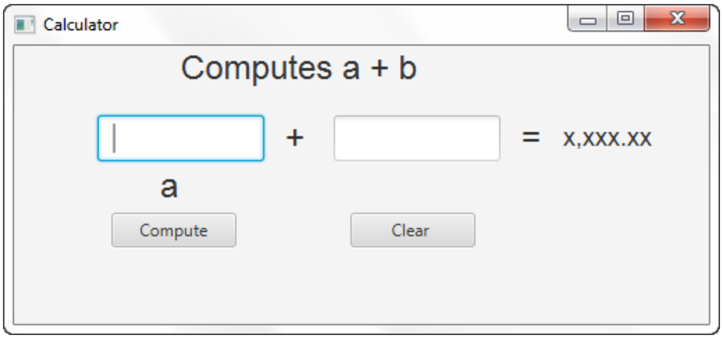


Figure 11.32
The application `AddingMachineV3`'s window.

11.6.2 `VBox` and `HBox` Layout Managers

A `VBox` layout manager is a `Pane` that places the components added to it in one vertical column. Within the column, the components are arranged from top to bottom in the order in which they were added to the container. An `HBox` layout manager is also a `Pane` that places the components added to it in one horizontal row. Within the row, the components are arranged from left to right in the order in which they were added to the container. If the `VBox` or the `HBox` pane is the only pane added to the scene, the scene's size is adjusted to display all of the components within it.

The spacing between all or individual components added to layout panes can be specified, as can the spacing between the edges of the pane container they are added to (the scene or another pane). In addition, if the container they are added to is larger than the space require to display them, their alignment within the container (e.g., center them) can be specified. Table 11.10 describes the constructors and commonly used methods in the `VBox` class. There are analogous methods in the `HBox` class.

Table 11.10
Constructors and Commonly Used Methods in the `VBox` Class

Method Constructors	Description * indicates the HBox class has an analogous methods
`VBox( )`	Constructs an empty VBox with no spacing between components*
`VBox(double spacingSize)`	Constructs an empty VBox with the specified `spacingSize` units between components*
`VBox(Node component, …)`	Constructs VBox with no spacing between components, the and adds the … `component` to it*
`VBox(double spacingSize, Node component, …)`	Constructs an VBox with the specified `spacingSize` units between components, and adds the … `component` to it*
Methods	
`void setPadding(new Insets(double units))`	Sets the spacing around the edges of the VBox to the specified `units` of spacing
`void setSpacing(double units)`	Sets the spacing between the VBox's components to specified `units` of spacing
`void setAlignment(Pos.A_CONSTANT)`	Sets the alignment of a VBox's components to the specified static constant defined in the class `Pos`
`void setMargin(Node component, new Insets(double units)`	Sets the spacing between a VBox and its adjacent components to the specified `units` of spacing

The code shown in Figure 11.33 creates the GUI window shown in Figure 11.34. Its scene is composed of a `VBox` pane, which contains an `HBox` and a `VBox` each containing three components.

Lines 1–6 create the six GUI components. The first three are added to an `HBox` named `hbPane` declared on line 8, and the second three are added to the `VBox` named `vbPane1` declared on line 9. Those lines also set the spacing between the three components they contain to be 5 units. Line 10 sets the maximum width of the object `vbPane1`'s rows to 216 units. Then these two layout panes are added to another `VBox` named `scenesPane` on line 12. The spacing between this pane's rows is set to 20 units on line 13, and the spacing around its outside edges is set to 10 units on line 14. Then this `VBox` is added to the application's scene on line 16.

```
1   Button btn1 = new Button("A Button");
2   TextField tf1 = new TextField("3 componets in an HBox");
3   Label lab1 = new Label("A Label");
4   Button btn2 = new Button("A Button");
5   TextField tf2 = new TextField("3 componets in a VBox");
```

```
Label lab2 = new Label("A Label");
HBox hbPane = new HBox(5, btn1, tf1, lab1);
VBox vbPane1 = new VBox(5, btn2, tf2, lab2);
vbPane1.setMaxWidth(216);
VBox scenesPane = new VBox(hbPane, vbPane1);
scenesPane.setSpacing(20);
scenesPane.setPadding(new Insets(10));
Scene scene = new Scene(scenesPane);
```

Figure 11.33
Part of the code used to create the window in shown in Figure 11.34.

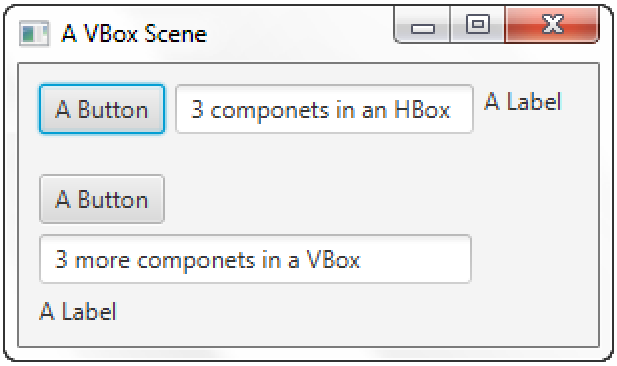


Figure 11.34
The window whose scene is created by the code in Figure 11.30.

11.6.3 FlowPane Layout Manager

A `FlowPane` is a `Pane` divided into equal width rows. Each row's width is the width of the pane. Each row's height is set to the height of the tallest component in the row. Figure 11.35 shows a GUI built using a `FlowLayout` pane. Five components were added to it: three buttons, then a label, and then a text field.

Beginning with the top row, components are positioned in the rows from left to right in the order in which they are added to the container. A row's height is adjusted to accommodate the component in the row whose height is the largest. The components in a row are left justified by default. When a row fills up, the next component is added to the row below it. If the pane's height cannot accommodate all the rows necessary to display the components, they are not shown or are partially shown. If a single component is too wide for a row, it is only partially displayed.

The application `FlowLayoutGUI`, shown in Figure 11.36, builds the graphical interface shown in Figure 11.35 using a flow layout manager. Following our GUI-building coding process, the components are declared and their properties are set on lines 21–28. The font of the component `second` is set, to a larger size than the default font size, on line 23.

The `FlowLayout` pane `root` is declared on lines 31–32 and all of the components are passed to its constructor. The first two optional arguments passed to the constructor set the spacing between components on the same row to 20 units, and the spacing between the rows to 10 units. Line 34 creates the application's scene, and the layout pane is passed to it, as are the width and height of the scene. That becomes the width and height of the flow layout pane because the preferred values of those dimensions were not specified.

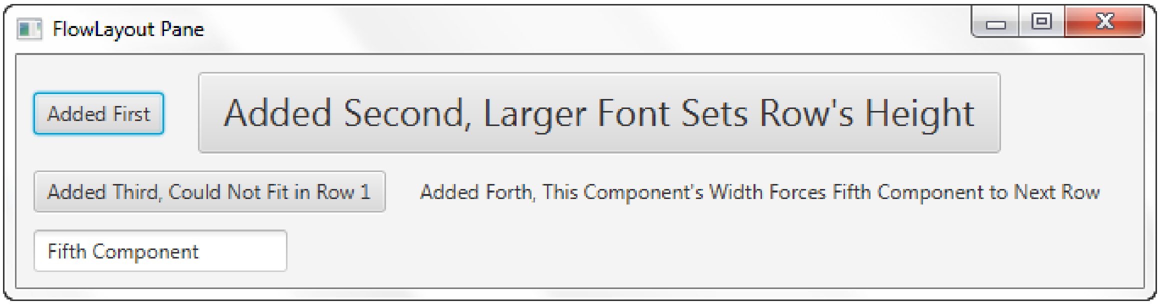


Figure 11.35
The application `FlowLayoutGUI's` window.

```
1   import javafx.application.Application;
2   import javafx.scene.Scene;
3   import javafx.scene.control.Button;
4   import javafx.scene.control.Label;
5   import javafx.scene.control.TextField;
6   import javafx.scene.layout.FlowPane;
7   import javafx.geometry.Insets;
8   import javafx.scene.text.Font;
9   import javafx.stage.Stage;
10
11  public class FlowLayoutGUI extends Application
12  {
13     Button first, second, third;
14     Label fourth;
15     TextField fifth;
16
17     @Override
18     public void start(Stage primaryStage)
19     {
20        //Steps 1 and 2: declare the components and specify their properties
21        first = new Button("Added First");
22        second = new Button("Added Second, Larger Font Sets Row's Height");
23        second.setFont(new Font("Sherif", 22));
24        third = new Button("Added Third, Could Not Fit in Row 1");
25        fourth = new Label("Added Forth, This Component's Width Forces " +
26                           "Fifth Component to Next Row" );
27        fifth = new TextField("Fifth Component");
```

```
        fifth.setPrefWidth(150);

        //Step 4: add the components to the container
        FlowPane root = new FlowPane(20, 10, first, second, third, fourth,
                                   fifth);

        Scene scene = new Scene(root, 600, 180);

        primaryStage.setTitle("FlowLayout Pane");
        primaryStage.setScene(scene);
        primaryStage.show();
     }

    public static void main(String[] args)
    {
       launch(args);
    }
 }
```

Figure 11.36
The application `FlowLayoutGUI`.

11.6.4 `BorderPane` Layout Manager

A `BorderPane` layout manager is a `Pane` that is divided into five regions named Top, Left, Center, Right, and Bottom. One component can be added to each region, which can be another pane containing multiple components. A component does not have to be added to a region. The positions of these regions in the border pane are shown in Figure 11.37.

The width of the Top and Bottom regions of a border pane is always the same, and the height of the Left, Center, and Right regions is always the same. The height of the Top and Bottom regions is adjusted to be the size of the components added to them.

When the size of the container the border pane is added to is not specified, the width of the Top and Bottom regions is adjusted to be the size of the widest component added to them. The height of the Left, Center, and Right regions is adjusted to be the largest of the heights of the components added to them.

When the size of the container the border pane is added to is specified, the width of the Top and Bottom regions is the container's specified width. The Center region's width is the width remaining between the Left and Right regions, and the height of these regions is the height between the Top and Bottom regions.

The `BorderPane` class has a no-parameter constructor and two other constructors as shown in the top of Table 11.11. The most often used one of these is the no-parameter constructor. Components can be added to the center region, or to all of the regions, when the `BorderPane` object is created using the two other constructors. More commonly, a region's component is added to it using the methods in the lower portion of the table.

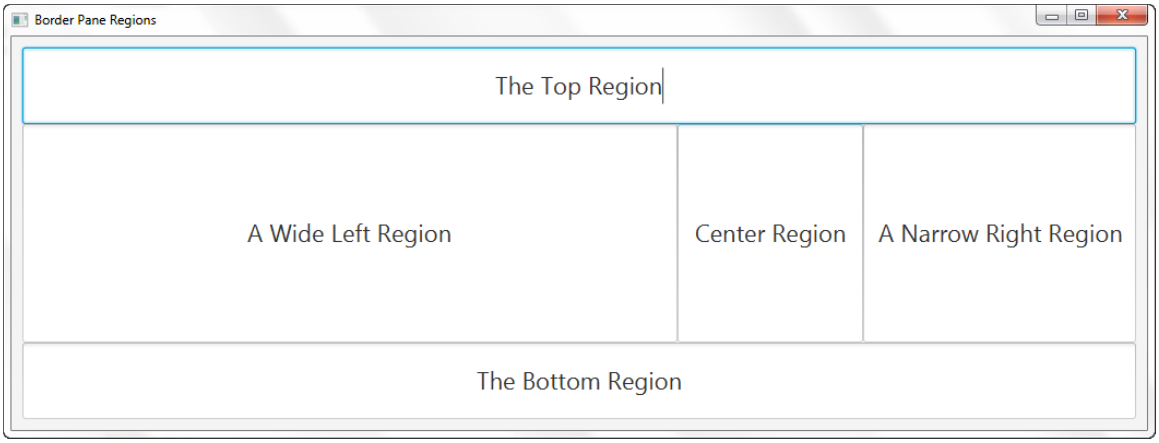


Figure 11.37
`BorderPane` layout regions.

Table 11.11
Constructors and Commonly Used Methods in the `BorderPane` Class

Method	Description
Constructors	
`BorderPane()`	Constructs and empty `Borderpane`
`BorderPane(Node center)`	Constructs a `BorderPane` with the component `center` in its Center region
`BorderPane(Node center, Node top, Node right, Node bottom, Node left`	Constructs a `BorderPane` with the components `center`, `top`, `right`, `bottom`, and `left` in the pane's Center, Top, Right, Bottom, and Left regions
Methods	
`void setTop(Node component)`	Places the `component` in the Top region of the `BorderPane`
`void setLeft(Node component)`	Places the `component` in the Left region of the `BorderPane`
`void setCenter(Node component)`	Places the `component` in the Center region of the `BorderPane`
`void setRight(Node component)`	Places the `component` in the Right region of the `BorderPane`
`void setBottom(Node component)`	Places the `component` in the Bottom region of the `BorderPane`
`void setPadding(new Insets(double value))`	Sets the spacing around the edges of the `BorderPane` to `value` units of spacing

The class `AddingMachineV4` shown in Figure 11.38 builds the GUI interface of the adding machine discussed in Section 11.4.1 using a `BorderPane` layout manager. The user interface it produces is shown in Figure 11.39. Although it is not identical to the interface shown in Figure 11.20 built by the class `AddingMachineV2` shown in Figure 11.19, the use of the border layout manager does facilitate the implementation of this GUI.

The GUI components are declared and their properties are set on Lines 19–38 of Figure 11.38. Then three `HBox` objects named `top`, `center`, and `bottom` are declared on lines 40, 42, and 45 and the components they will contain are passed to their constructors. Since the HBox objects `center` and `bottom` constructed on line 42 and 44 contain multiple components, their constructor is also passed a component horizontal separation of 10 and 50 units respectively.

The `BorderPane` object `root` is declared on Line 48, and the padding around its edges is set to 10 units on line 49. Then the three HBox objects `top`, `center`, and `bottom` are placed in `root`'s the Top, Center, and Bottom regions on lines 50–52. The `root` pane is passed to the application's scene on line 54.

```
1    import javafx.application.Application;
2    import javafx.geometry.*;
3    import javafx.scene.control.*;
4    import javafx.scene.layout.*;
5    import javafx.scene.text.Font;
6    import javafx.scene.Scene;
7    import javafx.stage.Stage;
8
9    public class AddingMachineV4 extends Application
10   {
11      Label description, plus, equals, sum, a, b;
12      TextField aValue, bValue;
13      Button compute, clear;
14
15      @Override
16      public void start(Stage primaryStage)
17      {
18        //Step 1: Declare the component objects, Step 2: Set their properties
19         description = new Label("Computes a + b");
20         description.setFont(new Font("Arial", 24));
21         aValue = new TextField();
22         aValue.setPrefSize(120, 30);
23         plus = new Label("+");
24         plus.setFont(new Font("Arial", 24));
25         bValue = new TextField();
26         bValue.setPrefSize(120, 30);
27         equals = new Label("=");
28         equals.setFont(new Font("Arial", 24));
```

```
        sum = new Label("x,xxx.xx");
        sum.setFont(new Font("Arial", 16));
        sum.setMinWidth(68);
        a = new Label("a");
        a.setFont(new Font("Arial", 24));
        b = new Label("b");
        b.setFont(new Font("Arial", 24));
        compute = new Button(" Compute ");
        clear = new Button  ("  Clear   ");
        Tooltip.install(clear, new Tooltip("Clears operands and result"));

        HBox top = new HBox(description);
        top.setAlignment(Pos.CENTER);
        HBox center  = new HBox(10, aValue, plus, bValue, equals, sum);
        center.setAlignment(Pos.CENTER);
        center.setPadding(new Insets(15));
        HBox bottom = new HBox(50, compute, clear);
        bottom.setAlignment(Pos.CENTER);

        BorderPane root = new BorderPane();
        root.setPadding(new Insets(10));
        root.setTop(top);
        root.setCenter(center);
        root.setBottom(bottom);

        Scene scene = new Scene(root);

        primaryStage.setTitle("BorderPane Layout Calculator");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }

}
```

Figure 11.38
The application `AddingMachineV4`.

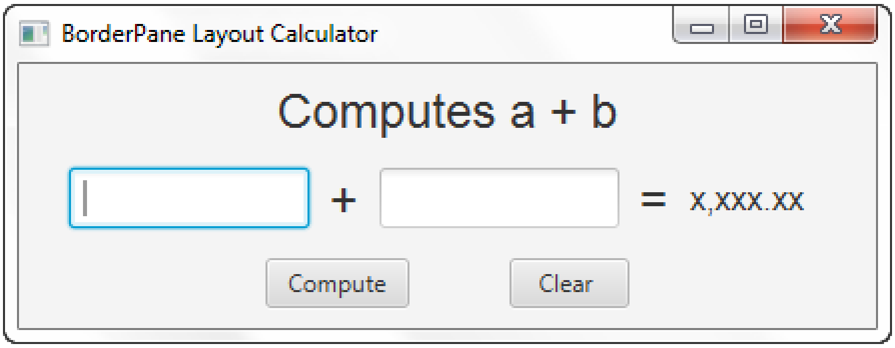


Figure 11.39
The GUI interface produce by the application `addingMachineV4`.

11.7 CHAPTER SUMMARY

User-friendly graphical interfaces are incorporated into a program by displaying GUI components such as buttons, text fields, and labels in a window that the user can interact with. Java FX is Java's third generation GUI package added to the Application Programmer Interface. It was preceded by Swing, which was preceded by AWT. Being third generation, FX provides features not available in Swing such as support for mobile touch devices, easier animation, and the use of special effects within a GUI.

The overloaded versions of the `showInputDialog` and `showMessageDialog` methods in the Swing's `JOptionPane` class can be used to display more informative and user-friendly dialog boxes than those versions used in previous chapters. Equivalent dialog boxes can be created in FX by declaring an instance of the `TextInput` or the `Alert` classes. Both the Swing and the FX implementations can tailor the displayed icon to the message being conveyed, and both can display a default input or a set of valid inputs from which to choose an input.

An FX application extends the class `Application`, and like other Java applications it must contain a method named `main` invoked by the runtime environment. Normally in an FX application, this method only contains an invocation of the method `launch`. The method `launch` creates a `Stage` object, the application's window, and then invokes the application's `start` method passing it the `Stage` object. GUI components are added to container objects that are added to a `Scene` object, and then the scene is added to the stage and the stage is displayed.

The (x, y) position of each GUI component added to a container, called a pane, can be specified by the programmer, or the program can use a layout manager to facilitate the positioning of the components. FX provides ten layout managers. Five of the more commonly used ones position the components in a row (`HBox`), a column (`VBox`), a set of rows or columns (`FlowPane`), a rectangular grid of cells (`GridPane`), or five predefined regions (`BorderPane`) within the container. A layout manager can be added to another layout manager.

An event handler is a method that is invoked to perform the application-dependent processing associated user interactions on the GUI components, or some other event such as dragging the mouse, or typing a keyboard key, or the end of a `KeyFrame` timer cycle. In FX that method's name is `handle`. It is the only abstract method defined in the functional interface `EventHandler`.

The application-dependent code is placed inside of an implementation of the method `handle` within the application, or defined in a Lambda expression that is passed to a method associated with the event. When invoked, the method `handle` is passed an object that contains information about the event that can be fetched by the application-dependent code using various get methods defined in the object's class.

There are two techniques used to add 2-D shapes, including lines, to a program's window. When the 2-D objects are instances of a class that inherits from the class `Shape`, such as the class `Circle`, we can use the same four-step approach used to add control objects (e.g., buttons, labels, etc.) to the container.

The second technique employs an art studio analogy. An instance of the class `Canvas` is created, and then a `GraphicsContext` brush object is created by invoking the method `getGraphicsContext2D` on the canvas. The shapes are drawn on the canvas by invoking 2-D shape methods defined in the `GraphicsContext2D` class on the brush object. After the shapes are drawn, the canvas is added to the window's container.

Knowledge Exercises

1. True or false:

a) GUI stands for Graphical Unified Interface.

b) GUIs do not reduce the time and effort needed to interact with a program.

c) To create a graphical user interface, we declare an instance of a `Pane` container class, add the GUI components to it, and then add it to the scene.

d) The GUI package added to Java most recently is FX.

e) Check boxes permit multiple selections to be made.

f) When a key is pressed an `Action` event occurs.

g) Java applications and Java FX applications both begin by executing in their `main` method.

h) A dialog box cannot be used to display several valid inputs from which a user can choose.

i) When the position of the GUI components is to be specified, a VBox container should be used.

j) The border layout manager specifies three regions for the components.

k) The North and South regions of the border layout are always the width of the scene.

l) All the cells of a grid layout must have the same height and width.

2. Give the FX code to ask a person for their age using a dialog box containing an exclamation point icon, and the title *Happy Birthday*.

3. Name at least three GUI components, and explain their usual uses.

4. Give the code to:

a) Declare a button named b1 containing the text “Click ME”.

b) Relocate `Button b2` to location (200, 400).

c) Fetch the text displayed in the `TextField` named `tf1` and output it to the system console.
d) Create a `TextField` named `tf2` whose default input is *First Name*.
e) Change the text of `Label lb1` to “The Answer is Yes”.
f) Add and the `Button b3` to the `Pane p1` and locate it at (200, 100).
g) Make the size of the `TextBox` named `tb2` 100 units wide and 50 units high.
h) Attach the tool tip “Click After Entering a Number” to the `Button` named `b4`.

5. Give the names, the regions of the border layout, and give the code to add a button to each region.
6. Describe the position of components c1, c2, and c3 added to a VBox layout manager.
   a) Give the code to make the pane `p2`’s layout manager `BorderLayout`.
7. Give the code to display a 600 x 800 unit red window from within an FX application.
8. What is the name of FX’s event handler method?
9. Give the Lambda expression to execute the method `clickHandler` that will determine which `Button` component was clicked.
10. Give the code to register the event handler coded in the class `MyClass` when the user types a non-control keyboard key.
11. Repeat question 10 for a left mouse click.
12. Three `Button` objects named `b1`, `b2`, and `b3` are registered with the same event handler. Give the code in the event handler method to output “Button 2” when `b2` is clicked.
13. Give the code to invoke the method `keyHandler()`when a key is released.
14. Give the code to draw a square on the `Canvas` object c1.

Programming Exercises

1. Write a program with a GUI that allows the user to input the length and width of a rectangle. The GUI will have three buttons. When one of the buttons is clicked, the area of the rectangle is displayed in square feet. When the second button is clicked, the perimeter of the rectangle is displayed. When the third button is clicked, the GUI is restored to the condition it was in when the program was launched. Each button will have its own event handler.
2. Modify the program in Exercise 1 so one event handler performs the calculations, and the restoration of the GUI to its launch condition is performed by separate methods.
3. Modify the program in Exercise 2 to include an additional input for the cost of carpet per square foot and a fourth button to calculate and display the cost of the rectangular carpet.
4. Create a program to display a GUI window with the following features: its size is 400 x 450 units. The window title is *Surprise!* The window should contain a button labeled “Press Here”. When the button is pressed, the background color should change to your favorite color and a circle (or a smiley face for extra credit) should appear.

5. Write a program whose GUI contains three buttons, labeled *RED*, *YELLOW*, and *BLUE*, which cause a filled rectangle to be drawn in the appropriate color when clicked.

6. Write a GUI application that will react to a mouse-click event by displaying the x and y coordinates of the position in the window where the mouse was clicked.

7. Create a GUI window with a mouse and a piece of cheese drawn on it, each at random locations. The keyboard directional arrow keys can be used to move the mouse to the piece of cheese.

8. Expand the application described in Exercise 7 so the user can also drag the cheese to the mouse.

9. Write a GUI application called `StopWatch` that displays the minutes and seconds that have elapsed since the interface's Start button was clicked. When the interface's Reset button is clicked, the elapsed time should return to zero, and the timer should stop. Each button should have separate, appropriately named, methods to perform their application-dependent processing.

10. Design and implement a GUI calculator that the user can use to add, subtract, multiply, or divide two operands. The operands will be entered into one text field by clicking buttons labeled 0 though 9, or one labeled with a decimal point. The input of the first operand will be terminated by the user clicking one of four buttons, each labeled with one of the four arithmetic operators. The input of the second operand entered will be terminated by the user clicking a button labeled with an equals sign. The result of performing the math operation on the two operands will be displayed in the text field. Also, include a Clear button to clear the calculator so another operation can be performed, and buttons to save, recall, and clear the contents of the text field.

11. Write a GUI application named *Know Your Shapes* that displays a colorful circle, square, rectangle, and ellipse, each with a text field below it. After the user types the names of each shape in the text boxes and clicks the Done button, display *Correct* in the text fields in which the typed names are correct and the correct name of the shape in the text boxes in which the typed name is incorrect. After five seconds have elapsed, clear the text boxes and output *Try again* to the GUI.

Enrichment

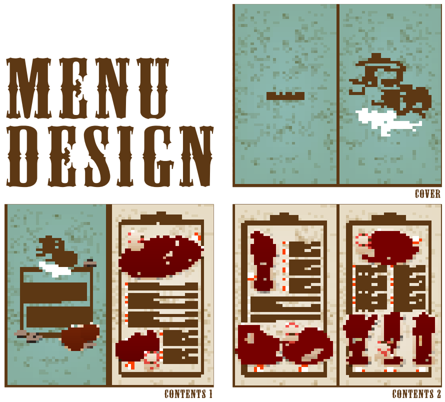
Create an FX program that displays a random animation of two graphical objects that bounce off the edges of the window and disappear when they collide.

Reference

Lowe, Doug. *FX for Dummies*. Hoboken NJ: John Wiley & Sons, 2015.

Chapter 12

Graphical User Interfaces: A Second Look



In this chapter

In Chapter 11, we became familiar with the techniques used to create an FX application's program window and add labels, buttons, and text fields to it, and how to respond to the user's interaction with these components via keystrokes or mouse clicks and drags. In this chapter, we expand our knowledge of the other GUI component classes available in the API FX package. Check boxes, radio buttons, combo boxes, and lists views allow a user to select one or more inputs from a set of valid inputs. The procedure for adding a component to a window is expanded to include the grouping of radio buttons and the use of collections.

Menus are a common component of GUIs, and both drop-down and pop-up menus are discussed in this chapter, as are submenus. In addition, the file-chooser dialog boxes and the color-picker component used to facilitate disk I/O and color selections are presented in this chapter.

After gaining a deeper understanding of the code we add to our programs to build a GUI, we will discuss the use of Scene Builder. Scene Builder is a Java provided GUI drag-and-drop interface incorporated into most of the popular integrated development environments, including NetBeans and Eclipse, which facilitates the creation of a graphical user interface.

After successfully completing this chapter, you will:

- Be able to display a border around GUI component groupings.
- Be able to create and position check boxes, radio buttons, combo boxes, and lists views and perform processing when the user selects inputs associated with these components.
- Understand how to implement drop-down and pop-up menus and submenus and perform processing in response to menu selections.

- Know how to use API defined dialog boxes to facilitate the input of file I/O paths and file names.
- Know how to use the color-picker component to select a predefined color or to compose a custom color.

12.1 BORDERS, CHECK BOXES AND RADIO BUTTONS

The GUI components check box and radio button are shown in the upper center and upper right portion of Figure 12.1. Groupings of check boxes and radio buttons are used to facilitate the selection of one or more inputs from a small set of valid inputs. When the user can select several of the inputs from the set, check boxes are used in the GUI. When the inputs are mutually exclusive, that is only one of the valid inputs can be selected, radio buttons are used.

Ordinarily a set of check boxes or a set of radio buttons is added to an instance of a Pane, most often a `VBox`, and then that is added to another pane or the application's scene. This makes the check box set or radio button set easier to reposition in the window because only the pane's location needs to be changed, rather than changing the position of each of the boxes or buttons individually. In addition, a label can be added to the top of the pane to provide additional information about the box or button set, and the pane's border can be made visible to give the visual impression that the check boxes or buttons are part of a set. This is the approach that was taken when the GUI illustrated in Figure 12.1 was created.

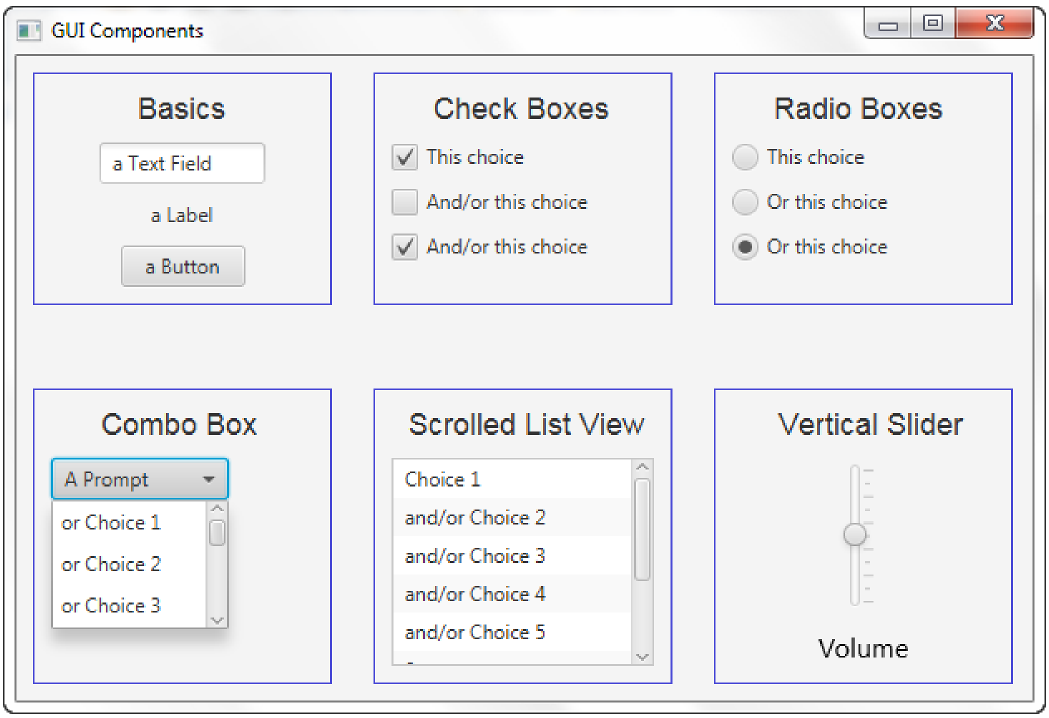


Figure 12.1
GUI components.

12.1.1 Pane Borders

One way of making a pane's border visible is to invoke the `setStyle` method, inherited from the class `Node`, on the pane object. The argument passed to it is a string that defines the CSS style of the object. In the API documentation, CSS stands for Cascading Style Sheets, which is an FX feature used to specify the visual appearance (style) of a window's individual components, its panes, or its entire scene.

The following lines of code were used to declare the `VBox` pane named `basicsPane` displayed in the upper left corner of Figure 12.1. The string argument used on the second line specifies the border color style of the pane to be blue, with an intensity of 70%.

```
VBox basicsPane = new VBox();
basicsPane.setStyle("-fx-border-color: rgba(0, 0, 200, 0.7);");
```

The style string passed to `setStyle` can be a concatenation of several style strings. For example, the code below also specifies the pane's background color to be red, with an intensity of 5%.

```
basicsPane.setStyle("-fx-border-color: rgba(0, 0, 200, 0.7);" +
                    "-fx-background-color: rgba(200, 0, 0, 0.05);";
```

The syntax used in FX CSS is similar to that used in HTML. In these examples the function `rgba()` stands for red-green-blue with alpha transparency. A detailed discussion of CSS is beyond the scope of this book.

12.1.2 Check Boxes

A grouping of check boxes is used on a GUI to facilitate the selection of one or more inputs from a small set of valid inputs. When the user clicks a check box, a check mark either appears in the check box or is removed from it. Using the techniques discussed in this section, check boxes are created, added to, and positioned in GUI containers. Processing can be initiated when the user checks or unchecks them. However, to give users time to review and finalized their selections, the processing associated with them is usually performed when a button within the scene is clicked.

Creating Check Boxes

Check boxes are instances of the FX class `CheckBox` and can be created using the class's one (`String`) parameter constructor, or its default constructor. The text that appears beside the check box when it is displayed is the string passed to the one-parameter constructor or the string passed to the class's `setText` method invoked on the component. The following code fragment creates the first two check boxes shown in the top center of Figure 12.1:

```
//Creates two check boxes and initializes their text
CheckBox cb1 = new CheckBox("This choice");
CheckBox cb2 = new CheckBox();
cb2.setText("And/or this choice");
```

The `setText` method can be used to set the text property of most GUI components that display text, and it can be used to change the text initially associated with a component. As discussed in

Chapter 11, the properties of components and containers can be set and fetched using the methods shown in Table 11.4 of that chapter, which is recreated in this chapter as Table 12.1 for convenience.

By default, a check box is initially displayed unchecked (without a check mark). When it is unchecked, the check box's value of its `selected` property is `false`. The method `setSelected` can be invoked on a check box object to change the value of the property to the Boolean value passed to it. The following two lines of code create a check box that contains a check mark.

```
//Creates a selected check box
CheckBox cb1 = new CheckBox("This choice");
cb1.setSelected(true);
```

Table 12.1
Methods Used to Specify a Component's Properties and Add it to a Container

Method Signature **Methods Invoked on Components**	**Description**
`setText(String theText)`	Changes the text displayed on the component to `theText`
`String getText()`	Returns the text displayed on the component
`setFont(Font fontStyle)`	Sets the font style of the container or component that invoked the method to `fontStyle`
`setVisible(Boolean visible)`	The component is made visible when the method is passed `true`, invisible when passed `false`
`setMinWidth(int mWidth)`	Sets the minimum width of a component to `mWidth`
`setMinfHeight(int mHeight)`	Sets the minimum height of a component to `mHeight`
`setStyle("-fx-background-color: pink")`	Sets the background color of a container to pink
`setLayoutX(double x)` `setLayoutY(double y)`	Sets the component's location within an instance of a `Pane` to (`x`, `y`)
Methods to Add Components to Containers	
`add(ParameterList)` `addAll(ParameterList)`	Adds a component (or components) to a container. The type of the object they are invoked on, and the parameter lists, depend on the container

Adding Check Boxes to Containers and Positioning Them

Like other components, check boxes are added to, and positioned in a GUI layout pane container using the techniques discussed in Chapter 11 Section 11.4.3 and Sections 11.6.2–11.6.4. The following code fragment adds three check boxes to the `VBox` container `checkPane`:

```
//Adds three check boxes to the VBox checkPane
CheckBox cb1 = new CheckBox("This choice");
CheckBox cb2 = new CheckBox("And/or this choice");
CheckBox cb3 = new CheckBox("And/or this choice");
VBox checkPane = new VBox(cb1, cb2, cb3);
```

They can also be added to, and positioned in, a pane container using the techniques discussed in Section 11.6.1.

Determining a Check Box's Status

The status of a check box, checked or unchecked, can be determined by invoking the `isSelected` method on it. The method returns the Boolean value `true` if the box is checked when the method is invoked, otherwise, it returns `false`. The following code fragment outputs *check box 1 was checked* to the system console.

```
//Determines if the checkbox cb1 is checked
CheckBox cb1 = new CheckBox("This choice");
cb1.setSelected(true);
if(cb1.isSelected() == true)
{  System.out.println("check box 1 was checked");  }
```

Check Box Events

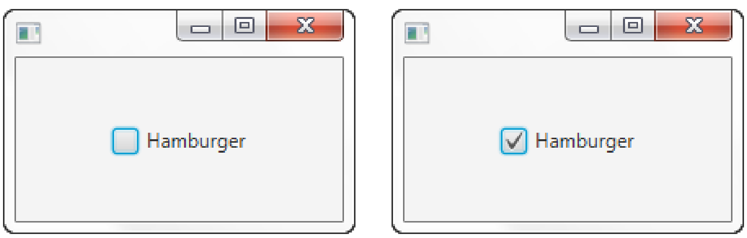
In most applications that use check boxes, the interface contains a button that is clicked after the user checks one or more of the check boxes, and then the processing associated with the checked boxes is performed from within the button's overridden version of the method `handle`. When this is the case, the status of the check boxes is determined by invoking the `isSelected` method within the code of the event handler method. The coding of the event handler and the techniques used to register it in the button's event handler list, are the same as those discussed in Section 11.5.1.

In applications where it is important to perform some processing *immediately after* a check box is clicked, a check box event handler is implemented. A check box click generates an `ActionEvent` and so the techniques used to perform the application-dependent processing when the event occurs is similar to those of a button click. The only difference is that the `setOnAction` method would be invoked on the `CheckBox` object.

Figure 12.2 shows the application `CheckBoxEvent` who's GUI interface, shown in Figure 12.3, contains one check box that is declared on line 14 and added to the applications container on line 18. The user can use the check box to select or unselect *Hamburger*. Its event handler code is defined within the Lambda expression on line 15. It contains one statement that invokes the method `cb1Handler` defined on lines 25–35. The Boolean condition of the `if` statement on line 27 of that method invokes the `isSelected` method on the check box object to determine if the click selected a hamburger. If so, line 29 performs the output. If the click unselected a hamburger, line 33 performs the output.

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class CheckBoxEvent extends Application
{
   CheckBox cb1;

   @Override
   public void start(Stage primaryStage)
   {
      cb1 = new CheckBox("Hamburger");
      cb1.setOnAction(e -> cb1Handler());

      StackPane root = new StackPane();
      root.getChildren().add(cb1);
      Scene scene = new Scene(root, 200, 100);

      primaryStage.setScene(scene);
      primaryStage.show();
   }

   public void cb1Handler()
   {
      if(cb1.isSelected() == true)
      {
         System.out.println("Hamburger Selected");
      }
      else
      {
         System.out.println("Hamburger Un-Selected");
      }
   }
}
```

Figure 12.2
The application `CheckBoxEvent`.



(a) (b)

Figure 12.3
The application `CheckBoxEvent`'s window.

12.1.3 Radio Buttons

Ordinarily, a radio button is grouped with other radio buttons into a mutually exclusive grouping because their most common use in GUIs is to facilitate the selection of *one* input from a small set of valid inputs. When the user clicks a radio button, a dot either appears on the button or is removed from it. Using the techniques discussed in this section, radio buttons can be created, added to, and positioned in GUI containers and made mutually exclusive.

Application-dependent processing can be initiated when the user selects, or unselects, a radio button. However, to give users time to review and finalize their selection, the processing associated with them is usually performed when a button within the scene is clicked.

Creating Radio Buttons

A radio button is an instance of the FX class `RadioButton`, and can be created using the class's one (`String`) parameter constructor or its default constructor. The text that appears beside the radio button when it is displayed is the string passed to its one-parameter constructor, or the string passed to its class's `setText` method invoked on the radio button. The below code fragment creates the first two radio buttons shown in the top-right portion of Figure 12.1.

```
//Creates two radio buttons and initializes their text
RadioButton rb1 = new RadioButton("This choice");
RadioButton rb2 = new RadioButton();
rb2.setText("Or this choice");
```

The `setText` method can be used to set the text property of most GUI components that display text, and it can be used to change the text initially associated with a component. As previously mentioned in this chapter, the properties of components and containers can be set and fetched using the methods shown in Table 12.1

By default, a radio button is initially displayed unselected without a center dot and its selected property is `false`. The method `setSelected` can be invoked on a check box object to change the value of that property to the Boolean value passed to it. The following two lines of code create a radio button that contains a center dot.

```
RadioButton rb1 = new RadioButton ("This choice");
rb1.setSelected(true);
```

Making Radio Buttons Mutually Exclusive

By default, a set of radio buttons is not mutually exclusive: one, several, or all of them could be selected at the same time. Because they are ordinarily used to choose one input from a set of mutually exclusive inputs, a set of radio buttons is normally designated to be mutually exclusive. When this designation is made, after one button in the set is selected, the previously selected button in the set is simultaneously unselected.

To designate a set of radio buttons to be mutually exclusive, their `ToggleGroup` property is set to the same `ToggleGroup` object. The following code fragment creates a mutually exclusive set of two radio buttons:

```
//Makes three radio buttons mutually exclusive
RadioButton white= new RadioButton("White Walls");
RadioButton yellow = new RadioButton("Yellow Walls");
ToggleGroup wallsColorGroup = new ToggleGroup();
white.setToggleGroup(wallsColorGroup);
yellow.setToggleGroup(wallsColorGroup);
```

Adding Radio Buttons to Containers and Positioning Them

Like other components, radio buttons are added to, and positioned in, a GUI layout pane container using the techniques discussed in Chapter 11 Section 11.4.3 and Section 11.6. The following code fragment adds three radio buttons to the `VBox` container `radioPane`:

```
//Adds three radio buttons to the VBox radioPane
RadioButton rb1 = new RadioButton("This choice");
RadioButton rb2 = new RadioButton ("Or this choice");
RadioButton rb3 = new RadioButton ("Or this choice");
VBox radioPane = new VBox(rb1, rb2, rb3);
```

The techniques used to add, and position, radio buttons in a pane container are discussed in section 11.6.1.

Determining a Radio Button's Status

The status of a radio button, selected or unselected, can be determined by invoking the `isSlected` method on it. The method returns the Boolean value `true` if the button is selected when the method is invoked, otherwise, it returns `false`. The following code fragment outputs *Hamburger was selected* to the system console.

```
//Determines if the radio button rb1 is selected
RadioButton rb1 = new RadioButton("Hamburger");
rb1.setSelected(true);
if(rb1.isSeclected() == true)
{
   System.out.println(rbi.getText() + "Hamburger was selected");
}
```

Radio Box Events

In most applications that use radio buttons, the interface contains a `Button` object that is clicked after the user selects one of the radio buttons in a group. Then the processing associated with the selection is performed from within the button's overridden version of the method `handle`. When this is the case, the status of a radio button in a group is determined by invoking the `is-Selected` method on the radio button within the code of the event handler. The coding of this method, and the techniques used to register it in the button's event handler list, are the same as those discussed in Section 11.5.1.

In applications where it is important to perform some processing *immediately after* a radio button is selected or unselected, a radio button event handler is implemented. Selecting a radio button generates an `ActionEvent` and so the technique used to perform the application-dependent process when the event occurs is similar to that of a button click. The only difference is that the `setOnAction` method would be invoked on the `RadioButton` object.

The application `MealMenu` shown in Figure 12.4 accepts a user's food order, and outputs the order to the system console. When the program is launched, the window in Figure 12.5a appears. Under the assumption that most customers would want napkins, the Napkins check box is initially checked. Figure 12.5b shows the window with a completed food order. The Place Order button is then clicked, and the order is output to the system console. The output for the order shown in Figure 12.5b would be *Taco, and cheese, and napkins*.

Line 21 of the application invokes the `setOnAction` method on the `Button` object `placeOrder` declared on line 20, and the invocation of the method `placeOrderHandler` in its Lambda expression becomes the code of the button's event handler. When the button is clicked, that method coded on lines 71–99 is invoked.

Lines 24–28 declare the label whose text is *Entrée* and the three radio buttons that make up the entrée part of menu, and they are then added to the `VBox` pane declared on line 30. Similarly lines 42–49 build the `VBox` layout pane that contains the check boxes preceded by a label whose text is *Extras*. Line 47 sets the napkins check box selected property to `true` causing it to be displayed with a check in it when the program is launched.

The radio buttons are made mutually exclusive on lines 36–39 by setting their `ToggleGroup` property to reference the same `ToggleGroup` object, `entreeGroup`, declared on line on 36. Lines 56–62 declare a `GridPane` layout manager, and place the two `Vbox` panes in its first row (line 60) and then place the Place Order button in its second row (line 61). The forth argument passed to the `add` method on line 61 specifies that the button will occupy two columns, and line 62 specifies that the button will be centered within the columns it occupies.

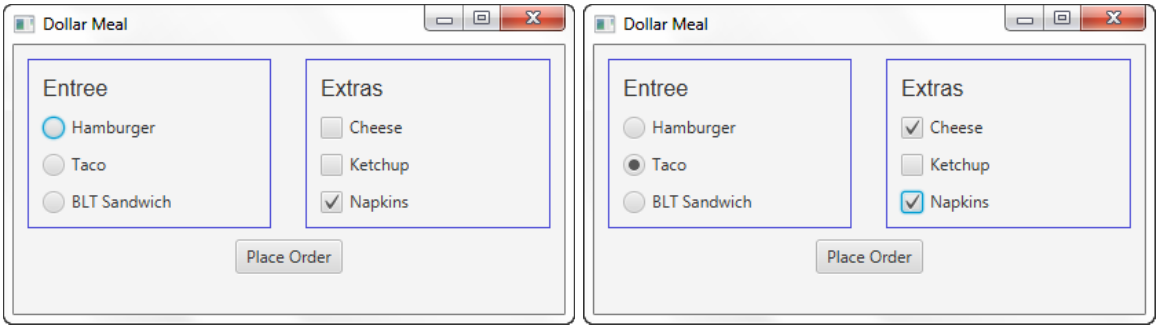
The decision statements in the method `placeOrderHandler` on lines 74–97, invoked when the Place Order button is clicked, invoke the `isSelected` method on the radio and check box objects to determine which items were selected or checked. If they were selected or checked the method returns `true`, and the item is added to the string `order` output on line 98.

```
1    import javafx.application.Application;
2    import javafx.geometry.*;
3    import javafx.scene.Scene;
4    import javafx.scene.control.*;
5    import javafx.scene.layout.*;
6    import javafx.scene.text.Font;
7    import javafx.stage.Stage;
8
9    public class MealMenu extends Application
10   {
11      RadioButton hamburger, taco, blt;
12      CheckBox cheese, ketchup, napkins;
13
```

```
   @Override
   public void start(Stage primaryStage)
   {
      String blue = "-fx-border-color: rgba(0, 0, 200, 0.7);";

      // Declare the order button and register its event handler
      Button placeOrder = new Button("Place Order");
      placeOrder.setOnAction(e -> placeOrderHandler());

      //Build the radio button VBox and group the buttons
      Label radioLbl = new Label("Entree");
      radioLbl.setFont(new Font("Arial", 16));
      hamburger = new RadioButton("Hamburger");
      taco = new RadioButton("Taco");
      blt = new RadioButton("BLT Sandwich");

      VBox radioPane = new VBox(radioLbl, hamburger, taco, blt);
      radioPane.setPrefWidth(180);
      radioPane.setStyle(blue);
      radioPane.setSpacing(10);
      radioPane.setPadding(new Insets(10));

      ToggleGroup entreeGroup = new ToggleGroup();
      hamburger.setToggleGroup(entreeGroup);
      taco.setToggleGroup(entreeGroup);
      blt.setToggleGroup(entreeGroup);

      //build the check box VBox
      Label extrasLbl = new Label("Extras");
      extrasLbl.setFont(new Font("Arial", 16));
      cheese = new CheckBox("Cheese");
      ketchup = new CheckBox("Ketchup");
      napkins = new CheckBox("Napkins");
      napkins.setSelected(true);

      VBox checkPane = new VBox(extrasLbl, cheese, ketchup, napkins);
      checkPane.setPrefWidth(180);
      checkPane.setStyle(blue);
      checkPane.setSpacing(10);
      checkPane.setPadding(new Insets(10));

      //Add the VBoxes and the button to a grid pane
      GridPane grid = new GridPane();
      grid.setPadding(new Insets(10));
      grid.setHgap(25);
      grid.setVgap(8);
      grid.addRow(0, radioPane, checkPane);
      grid.add(placeOrder, 0, 1, 2, 1);
      grid.setHalignment(placeOrder, HPos.CENTER);
```

```

        Scene scene = new Scene(grid, 400, 195);

        primaryStage.setTitle("Dollar Meal");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void placeOrderHandler()
    {
        String order = "";
        if(hamburger.isSelected())
        {
            order = order + "Hamburger";
        }
        else if(taco.isSelected())
        {
            order = order + "Taco";
        }
        else if(blt.isSelected())
        {
            order = order + "BLT sandwich";
        }
        if(cheese.isSelected())
        {
            order = order + ", and cheese";
        }
        if(ketchup.isSelected())
        {
            order = order + ", and ketchup";
        }
        if(napkins.isSelected())
        {
            order = order + ", and napkins";
        }
        System.out.println(order);
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 12.4
The application `MealMenu`.



(a) (b)

Figure 12.5
The application `MealMenu`'s window before and after the selections have been finalized.

The output for 12.5 b is: *Your order is Taco, and cheese, and napkins*

12.2 COMBO BOXES AND LIST VIEWS

The GUI components `ComboBox` and `ListView` are shown in the lower left and lower center portion of Figure 12.1. These components are similar to a set of radio buttons and a set of check boxes in that they are used to facilitate the selection of one or more inputs from a set of valid inputs items. When the number of items in the set is small, most GUI designers use radio buttons and check boxes to present the selection alternatives. Otherwise a combo box or a list view is the preferred component, because they can include a scroll bar to permit the user to view the alternative selections without taking up a large portion of the program's window.

By default, the number of rows of items displayed in these two components is dependent on the height of the pane they are added to. When the pane's height not large enough to display all of the items, a scroll bar is displayed. Alternately, the number of items displayed in a combo box's drop-down panel can also be specified by invoking the `setVisibleRowCount` method on the combo box object and passing it an integer row count.

A combo box is analogous to a set of radio buttons in that the user can only select one of its items, and a list view is analogous to a set of check boxes in that the user can select one or more of its items. However, by default only one item can be selected from a list view's items. To permit the user to select multiple items from a list view, the `setSelectionMode` method is invoked on the list view's data member `SelectionModel` and passed the `SelectionMode` class's static constant `MULTIPLE` as shown below.

```
ListView aListView = new ListView();
aListView.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

To select multiple sequential items from a list view's selections the user holds down the Shift key and clicks the first and last items in the sequential grouping. To select a group of non-sequential items, the Ctrl key is held down while the items are clicked.

Another difference between a combo box and a list box is that a combo box can display a prompt, or display a text field that the user can type a selection into that is not in the combo box's items list. By default, a prompt and the text field are not shown. Table 12.2 presents commonly used methods to construct a combo box and a list view, sets their properties, adds items to them, and fetches the selected item(s). We will discuss all of these methods within this section.

Table 12.2
Methods to Construct a Combo Box and a List View, Set Their Properties, and Add Items to Them

Method	Description
Constructors	
`ComboBox()` `ListView()`	Constructs objects that contains no items
`ComboBox(ObservableList <T> items)` `ListView(ObservableList <T> items)`	Constructs objects that contain the items in the `ObservableList` object passed to it
Methods in the `ComboBox` class	
`void setPromptText(String thePrompt)`	Displays the argument `thePrompt` passed to it on the combo box
`void setEditable(boolean value);`	Makes a `ComboBox` object's `textField` visible when `value` is **`true`**, and accepts the user entry as a valid item choice
`void setVisibleRowCount(int nRows)`	Displays a maximum of `nRows` items when the `ComboBox`'s drop-down arrow is clicked
`getValue()`	Returns the selected item whose type is <T>
Methods in the `ListView` class	
`getSelectedModel().setSelectedMode(SelectedMode.MULTIPLE)`	Permits multiple item selections. Use the static constant `SINGLE` to permit one item selection
`getSelectionModel().getSelectedItems()`	Returns an `ObservableList` object containing the items selected.
Method in the `ComboBox` and `List` classes	
`ObservableList<T> getItems()`	Fetches the address of the `ObservableList` containing the component's items
Methods in the `ObservableList` class	
`add(ItemType item)` `add(int index, ItemType item)`	Adds the item to the end of the list Adds the item at the specified index in the list
`addAll(ItemType item1, item2, ...)`	Adds all the items to the end of the list
Static method in the `FXCollections` class	
`ObservableList observableArrayList(ItemType item1, item2, ...)`	Creates and returns an `ObservableList` object that contains the items passed to it

Adding Items to a Combo Box and List View

A `ComboBox` and a `ListView` object contain a data member named `items` that references the list of valid selection items it can display. More specifically, the data member stores the address of an `ObservableList` object, whose class name reflects the fact that the list of items can be *observed* by the user. There are two techniques used to add items to an observable list. The first one will be explained using an instance of a `ComboBox`, and the second one will be explained using an instance of a `ListView`.

In the first of these techniques, after creating a `ComboBox` object, we fetch the address of its `ObservableList` object by invoking the class's `getItems` method on it. Then the `addAll` method is invoked on the returned object and the combo box's selection items are passed to it as arguments. The following code segment declares a combo box named `entrees`, then fetches the address of the available list stored in its data member `items`, and then adds the items *Hamburger* and *Taco* to its `ObservableList` of items.

```
//First technique for adding items to a combo box or list view
ComboBox <String> entrees = new ComboBox();
ObservableList <String> entreesItemList = entrees.getItems();
entreesItemList.addAll("Hamburger", "Taco");
```

An equivalent, and slightly more succinct, coding is:

```
ComboBox <String> entrees = new ComboBox();
entrees.getItems().addAll("Hamburger", "Taco");
```

The version of the `addAll` method invoked above, always adds the items passed to it to the end of the list in the same order as the order of the items passed to it. Alternately, the method `add` can be used to append one item to the list or add an item at a specified position (index) in the list. The below code adds the string *BLT Sandwich* to the beginning of the items list of the combo box named `entrees`.

```
//Adding one item to the combo box entrees at a specified location
entrees.getItems().add(0, "BLT Sandwich");
```

The second technique used to add items to either a combo box or a list view, is to create an instance of the `ObservableList` class containing the items, and pass that object's address to the constructor when the combo box or list view object is created. After creating the combo box or list view, the constructor stores the address passed to it in the object's `items` data member. The `FXCollections` class's static method `observableArrayList` is used to create the `ObservableList` object. The following code sequence creates a list view `extras`, containing the items *Fork* and *Napkins*.

```
//Second technique for adding items to a combo box or list view
ObservableList extrasItems = FXCollections.observableArrayList("Fork",
                                                                "Napkins");
ListView extras = new ListView(extrasItems);
```

Fetching the Selected Items

To fetch the item selected from a combo box the `getValue` method is invoked on the `ComboBox` object. The returned value is the item selected. If the `String` item Taco was selected from the combo box `entrees`, the below line of code would output *You selected a Taco* to the system console.

```
System.out.println("You selected a " + entrees.getValue());
```

To fetch the item, or items, selected from a list view is not as simple. First the `getSelectionModel` method is invoked on the `ListView` object. This method returns an instance of the class `MultipleSelectionModel`. Then the `getSelectedItems()` method is invoked on that object to fetch an `ObservableList` object that contains the items selected. The items selected can be gleaned out of that object using an enhanced for loop. The below code sequence outputs all of the items selected from the `ListView` object `extras` to the system console.

```
//Fetching the items selected from a list view
ObservableList<String> extrasSelected;
extrasSelected = extras.getSelectionModel().getSelectedItems();
for(String extra : extrasSelected)
{
   System.out.println(extra);
}
```

Combo Box and List View Events

In most applications that use a combo box and/or a list view, the interface contains a `Button` object that is clicked after the user finalizes the item selection(s). Then the processing associated with the selection is performed from within the button's overridden version of the method `handle`. The coding of this method, and the techniques used to register it in the button's event handler list, are the same as those discussed in Section 11.5.1.

In applications where it is important to perform some processing *immediately after* an item is selected, a combo box and/or a list view event handler is implemented. Selecting an item generates an `ActionEvent`, and so the technique used to perform the application-dependent process when the selection occurs is similar to that of a button click. The only difference is that the `setOnAction` method would be invoked on the `ComboBox` or the `ListView` object.

Figure 12.6 presents the application `MealMenu2` that accepts a user's food order using a combo box and a list view, and outputs the order to the system console. When the program is launched, the window in Figure 12.7a appears. Figure 12.7b shows the window with a completed food order. The Place Order button is then clicked, and the order is output to the system console. The output for the order shown in Figure 12.5b would be: *Your order is a BLT Sandwich, and Mayonnaise, and a Paper Plate, and Utensils.*

Lines 26–29 declares the `ObservableList` object containing the entrée's items, which is passed to the `ComobBox` constructor on line 30. Line 31 places the label text *Make a Selection* within the combo box, and then adds it and its label to a `VBox` on line 32. Lines 41 declares the

ListView object extras, and lines 42–44 uses the alternate technique previously discussed to define and add items to it. Line 45 permits multiple selections to be made from the list box. Line 46 adds the List View and its label to a second VBox.

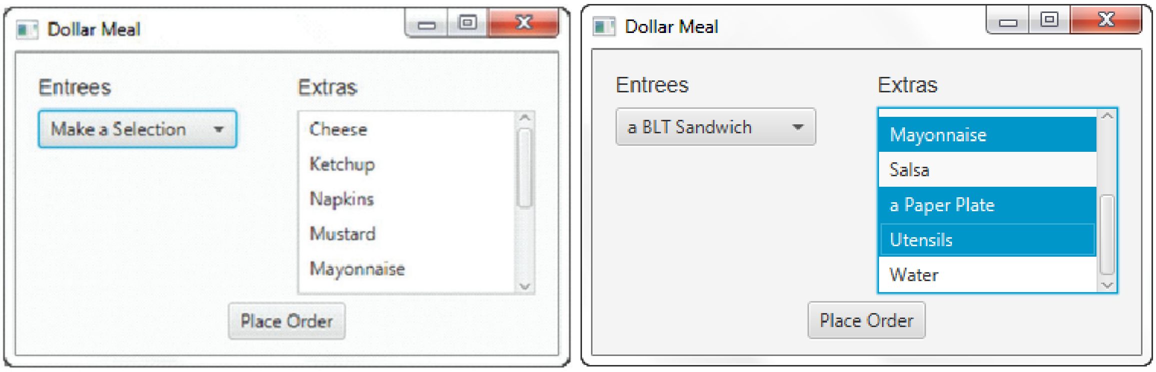
Line 53 of the application invokes the setOnAction method on the Button object declared on line 52, and the invocation of the method placeOrderHandler in its Lambda expression becomes the code of the button's event handler. When the button is clicked, the method coded on lines 66–76 is invoked. Line 68 adds the selected entrée item to the string output in line 75, and lines 69–74 add the selected extra items to it. Line 70 places the selected extra items into the ObservableList object declared on line 69, and then the enhanced for loop that begins on line 71 adds them to the output string.

```
1    import javafx.application.Application;
2    import javafx.collections.*;
3    import javafx.geometry.*;
4    import javafx.scene.control.*;
5    import javafx.scene.layout.*;
6    import javafx.scene.text.Font;
7    import javafx.scene.Scene;
8    import javafx.stage.Stage;
9
10   public class MealMenu2 extends Application
11   {
12      ComboBox entrees;
13      ListView extras;
14      Button placeOrder;
15
16      @Override
17      public void start(Stage primaryStage)
18      {
19         GridPane root = new GridPane();
20         root.setPadding(new Insets(10));
21         root.setHgap(5);
22
23         //Declare the ComboBox's label, its items, the ComboBox and VBox
24         Label entreesLbl = new Label("Entrees");
25         entreesLbl.setFont(new Font("Arial", 14));
26         ObservableList entreeItems = FXCollections.observableArrayList(
27                                  "a Taco", "a BLT Sandwich", "Nachos",
28                                  "a Hambuger", "Chicken Soup",
29                                  "Hot Chili", "a Salad");
30         entrees = new ComboBox(entreeItems);
31         entrees.setPromptText("Make a Selection");
32         VBox comboPane = new VBox(entreesLbl, entrees);
33         comboPane.setPrefWidth(180);
34         comboPane.setPrefHeight(180);
35         comboPane.setSpacing(7);
36         comboPane.setPadding(new Insets(5));
```

```

        //Declare the ListView's label, its items, the ListView and VBox
        Label extrasLbl = new Label("Extras");
        extrasLbl.setFont(new Font("Arial", 14));
        extras = new ListView();
        ObservableList<String> extrasList = extras.getItems();
        extrasList.addAll("Cheese", "Ketchup", "Napkins", "Mustard",
                          "Mayonnaise", "Salsa", "a Paper Plate",
        extras.getSelectionModel().setSelectionMode(SelectionMode.
                                                    MULTIPLE);
        VBox listPane = new VBox(extrasLbl, extras);
        listPane.setPrefWidth(180);
        listPane.setSpacing(7);
        listPane.setPadding(new Insets(5));

        //Declare the Place Order button and register its event handler
        placeOrder = new Button("Place Order");
        placeOrder.setOnAction(e -> placeOrderHandler());

        root.addRow(0, comboPane, listPane);
        root.setHalignment(placeOrder, HPos.CENTER);
        root.add(placeOrder, 0, 1, 2, 1);

        Scene scene = new Scene(root, 360, 200);

        primaryStage.setTitle("Dollar Meal");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void placeOrderHandler()
    {
        String order = "Your order is " + entrees.getValue();
        ObservableList<String> extrasSelected;
        extrasSelected = extras.getSelectionModel().getSelectedItems();
        for(String extra : extrasSelected)
        {
            order = order + ", and " + extra;
        }
        System.out.println(order);
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 12.6

The application `MealMenu2`.



(a) (b)

Figure 12.7
The application `MealMenu2`'s window.

The output is: *Your order is a BLT Sandwich, and Mayonnaise, and a Paper Plate, and Utensils*

12.3 MENUS

A menu is a GUI component used to obtain one of several valid inputs from the program user. In this way, menus are similar to combo boxes. The advantage menus have over combo boxes and the other GUI components we have discussed is that they can be used to present a wide variety of valid inputs while occupying a relatively small portion of the program's window. In addition, because many applications place menus in the same position within their window, they present an input interface that is familiar to the user.

The Java API supports two types of menus: drop-down menus and pop-up menus. The most common location for a drop-down menu is just below the window's title bar. Pop-up menus remain invisible until the user performs a platform-dependent action, usually a mouse action or key action. The most common mouse action to expose a pop-up menu is a right button mouse click. We will begin our discussion of menus with drop-down menus.

12.3.1 Drop-Down Menus

The program window in Figure 12.8a shows a drop-down menu whose menu bar is positioned just below the window's title bar. The menu-bar object contains one drop-down menu object on its left side that has the string "A Menu" associated with it. The user has clicked this menu object to expose the menu's five drop-down items and then clicked, or hovered over, the forth of these items to expose another dropdown menu containing three more items. Figure 12.8b gives the FX classes of the objects that make up the program's menu, displayed in Figure 12.8a. These classes are `MenuBar`, `Menu`, and `MenuItem`. The two submenu objects shown in Figure 12.8a are instances of the class `Menu`.

The `MenuItem` objects within a drop-down menu are the terminal components of the menu. These items are the set of valid inputs. The user selects one of these inputs by clicking it, which generates an action event just as clicking a `Button` object generates an action event. To display the menu bar just below the window's title bar, the menu bar is positioned at the top of the pane passed to the scene's constructor.

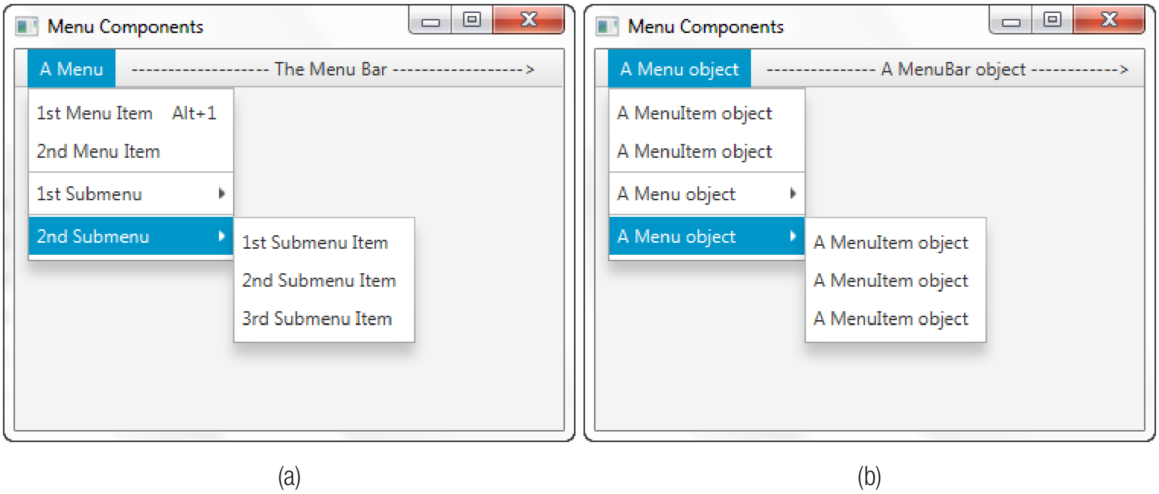


Figure 12.8
Drop-down menu components and their API classes.

Building a Drop-Down Menu System

Generally speaking, a drop-down menu system consisting of a menu bar containing one or more drop-down menus is created using the following six-step process. The below code fragments, used to illustrate each step of the process, were used to create the portion of a window's menu that is displayed in Figure 12.9.

1. Create the menu bar

```
MenuBar aMenuBar = new MenuBar();
```

2. Create the menus and their submenus

```
Menu dollarMenu = new Menu("Dollar");
Menu extrasMenu = new Menu("Extras");
Menu sandwichMenu = new Menu("Sandwich");
Menu mexicanMenu = new Menu("Mexican");
```

3. Create the menu items and the submenu items

```
MenuItem saladItem = new MenuItem("Salad");
MenuItem chickenSoupItem = new MenuItem("Chicken Soup");
MenuItem hamburgerItem = new MenuItem("Hamburger");
MenuItem bltItem = new MenuItem("BLT");
```

4. Add the items to the submenus and the items and submenus to the menus

```
sandwichMenu.getItems().addAll(hamburgerItem, bltItem);
dollarMenu.getItems().addAll(saladItem, chickenSoupItem,
                             sandwichMenu, mexicanMenu);
```

5. Add the menus to the menu bar

```
aMenuBar.getMenus().addAll(dollarMenu, extrasMenu);
```

6. Add the menu bar to the pane container

```
Pane pane = new Pane();
pane.getChildren().addAll(aMenuBar);
```

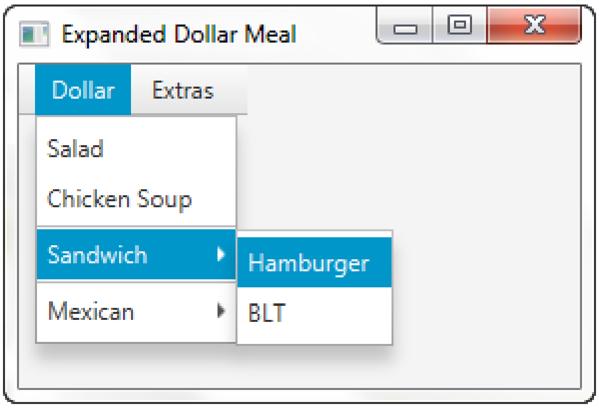


Figure 12.9
A drop-down menu system that contains two submenus.

Menu Separator Bars

Separator bars are added to a drop-down menu to visually group related elements, or to separate unrelated items. There are two separator bars in the Dollar drop-down menu shown in Figure 12.9: one above the Sandwich submenu and one below it. Separator bars are menu items that are instances of the class `SeparatorMenuItem`, and so they are added to a menu the same way other menu items are added to it. The two menu separators in Figure 12.9 were added to the drop-down menu `dollarMenu` using the following code fragment:

```
//Add a separator bar above and below the sandwich submenu
SeparatorMenuItem seperator1 = new SeparatorMenuItem();
SeparatorMenuItem seperator2 = new SeparatorMenuItem();
dollarMenu.getItems().addAll(saladItem, chickenSoupItem,separator1,
                             sandwichMenu, separator2 mexicanMenu);
```

Menu Item Events

When the user clicks one of the items in a menu, an action event occurs just as when a Button object is clicked. The processing associated with the selected menu item is performed within the overridden method handle, whose code can be defined in a Lambda expression passed to the `setOnAction` method invoked on the menu item. The below code would cause the method `dollarMenuItemHandler` to be invoked when the `MenuItem` object `saladItem` is clicked.

```
saladItem.setOnAction(e-> dollarMenuItemHandler(e));
```

Most often, clicks on any of the terminal items in a menu are handled by the same method. When this is the case, the `getSource` method can be invoked within the method on the `ActionEvent` object passed to it to identify which of the menu items was selected by the user. The Boolean condition in the below `if` statement is `true` if the user clicked the `MenuItem` object `saladItem`.

```
if( e.getSelected() ==  saladItem)
{
   //The processing associated with a saladItem selection
}
```

The application `ExpandedDollarMealMenu` shown in Figure 12.10 contains the code to build and display the menus shown in Figure 12.11 using our six-step process. The three Mexican food items displayed in the window on the right side of the figure are defined on lines 29–31, and added to the Mexican food submenu on line 45. Lines 46–47 add the items and submenus to the dollar meal drop-down menu, and also add the two separator bars created on lines 42–43 to it.

The application also contains the code to determine the user's meal selection, from the Dollar menu, and output it to the system console. After the seven terminal items are created within the third step of our process, lines 33–39 register the event handler for the seven terminal items (one item per line) by invoking the `setOnAction` method on them and passing the method a Lambda expression. Since all item clicks will be serviced by the method `dollarMenuItemHandler (ActionEvent e)` coded on lines 68–102, the Lambda expressions contain the same one line of code: an invocation of that method.

The Boolean expressions within `if-else` statements of the method `dollarMenuItemHandler` invoke the `getSource` method on the `ActionEvent` argument passed to the method (e.g., line 72) to determine the user's selection. This method returns the address of the `MenuItem` object that was clicked.

```
1    import javafx.application.Application;
2    import javafx.event.ActionEvent;
3    import javafx.scene.Scene;
4    import javafx.scene.control.*;
5    import javafx.scene.layout.Pane;
6    import javafx.stage.Stage;
7    public class ExpandedDollarMealMenu extends Application
8    {
9       MenuItem saladItem, chickenSoupItem, hamburgerItem, bltItem,
10               tacoItem, nachoItem, chiliItem;
11
12      @Override
13      public void start(Stage primaryStage)
```

```
    {
        //1- Create the menu bar, which is an instance of the class
        MenuBar aMenuBar = new MenuBar();

        //2- Create the menus and their submenus
        Menu dollarMenu = new Menu("Dollar");
        Menu extrasMenu = new Menu("Extras");
        Menu sandwichMenu = new Menu("Sandwich");
        Menu mexicanMenu = new Menu("Mexican");

        //3-Create menu items and sub-menu items. Register event handlers
        saladItem = new MenuItem("Salad");
        chickenSoupItem = new MenuItem("Chicken Soup");
        hamburgerItem = new MenuItem("Hamburger");
        bltItem = new MenuItem("BLT");
        tacoItem = new MenuItem("Taco");
        nachoItem = new MenuItem("Nacho");
        chiliItem = new MenuItem("Chili");

        saladItem.setOnAction(e-> dollarMenuItemHandler(e));
        chickenSoupItem.setOnAction(e-> dollarMenuItemHandler(e));
        hamburgerItem.setOnAction(e-> dollarMenuItemHandler(e));
        bltItem.setOnAction(e-> dollarMenuItemHandler(e));
        tacoItem.setOnAction(e-> dollarMenuItemHandler(e));
        nachoItem.setOnAction(e-> dollarMenuItemHandler(e));
        chiliItem.setOnAction(e-> dollarMenuItemHandler(e));

        //4- Add items to the submenus and items and submenus to the menus
        SeparatorMenuItem separator1 = new SeparatorMenuItem();
        SeparatorMenuItem separator2 = new SeparatorMenuItem();
        sandwichMenu.getItems().addAll(hamburgerItem, bltItem);
        mexicanMenu.getItems().addAll(tacoItem, nachoItem, chiliItem);
        dollarMenu.getItems().addAll(saladItem, chickenSoupItem, separator1,
                                     sandwichMenu, separator2, mexicanMenu);

        //5- Add the menus to the menu bar
        aMenuBar.getMenus().addAll(dollarMenu, extrasMenu);

        //6- Add the menu bar to the pane container
        Pane pane = new Pane();
        pane.getChildren().addAll(aMenuBar);

        Scene scene = new Scene(pane, 350, 200);

        primaryStage.setTitle("Expanded Dollar Meal Menu");
        primaryStage.setScene(scene);
```

```
      primaryStage.show();
   }

   public static void main(String[] args)
   {
      launch(args);
   }

   public void dollarMenuItemHandler(ActionEvent e)
   {
      String order = "You ordered ";

      if(e.getSource() == saladItem)
      {
         order = order + "a Salad";
      }
      else if(e.getSource() == chickenSoupItem)
      {
         order = order + "Chicken Soup";
      }
      else if(e.getSource() == bltItem)
      {
         order = order + "a BLT Sandwich";
      }
      else if(e.getSource() == hamburgerItem)
      {
         order = order + "a Hamburger";
      }
      else if(e.getSource() == tacoItem)
      }
         order = order + "a Taco";
      }
      else if(e.getSource() == nachoItem)
      {
         order = order + "a Nacho";
      }
      else if(e.getSource() == chiliItem)
      {
         order = order + "Chili";
      }

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

Figure 12.10
The application ExpandedDollarMealMenu.

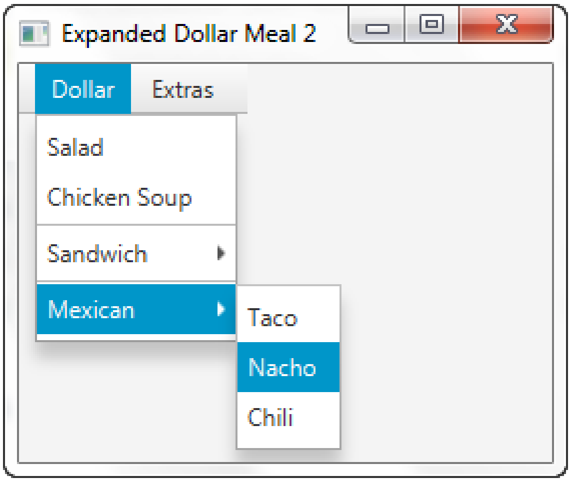


Figure 12.11
Five of the terminal nodes of the `ExpandedDollarMealMenu` application's Dollar menu.

Check Menu and Radio Menu Items

Check menu and radio menu items permit the user to make more than one selection from a drop-down menu. As discussed in Section 12.1.3, most often a radio item is grouped with other radio items, and only one selection can be made from within the grouping. Like the other menu items, check menu and radio menu items can be added to a menu displayed on the menu bar, or added to its submenus. Unlike the radio buttons discussed in Section 12.1.3, a radio menu item is displayed with a box to its left rather than a circle.

A check menu item is an instance of the class `CheckMenuItem`, and a radio menu item is an instance of the class `RadioMenuItem`. The techniques used to declare them, add them to menus and submenus, and service a click event on them, are the same techniques used for the menu items previously discussed in this section. In addition, the technique used to group radio menu items into a mutually exclusive toggle group is the same technique used to group radio buttons discussed in Section 12.1.3.

The following code sequence creates three check menu items, and adds them to the `Menu` object `extrasMenu`, and also adds the separator `separator1` and the submenu `beverageMenu` to the extras menu (see Figure 12.12b).

```
Menu beverageMenu = new Menu("Beverage");
SeparatorMenuItem separator1 = new SeparatorMenuItem();

CheckMenuItem cheeseItem = new CheckMenuItem("Cheese");
CheckMenuItem ketchupItem = new CheckMenuItem("Ketchup");
CheckMenuItem napkinsItem = new CheckMenuItem("Napkins");

extrasMenu.getItems().addAll(cheeseItem, ketchupItem, napkinsItem,
                             separator1, beverageMenu);
```

The following code creates the Beverage submenu's items shown in Figure 12.12b. It declares three radio menu items, makes them mutually exclusive, and adds them to the submenu `beverageMenu`.

```
RadioMenuItem waterItem = new RadioMenuItem("Water");
RadioMenuItem juiceItem = new RadioMenuItem("Juice");
RadioMenuItem sodaItem = new RadioMenuItem("Soda");
ToggleGroup beverageGroup = new ToggleGroup();
waterItem.setToggleGroup(beverageGroup);
juiceItem.setToggleGroup(beverageGroup);
sodaItem.setToggleGroup(beverageGroup);

beverageMenu.getItems().addAll(waterItem, juiceItem, sodaItem);
```

Adding all of the above sections of code to the application `ExpandedDollarMealMenu` shown in Figure 12.10 produces the drop-down menu and submenu shown in Figure 12.12. Figure 12.12a shows the application's window, and Figure 12.12b shows the window after the user has clicked the Extras menu and made two selections from its three check items, and one selection from the Beverage submenu's radio items.

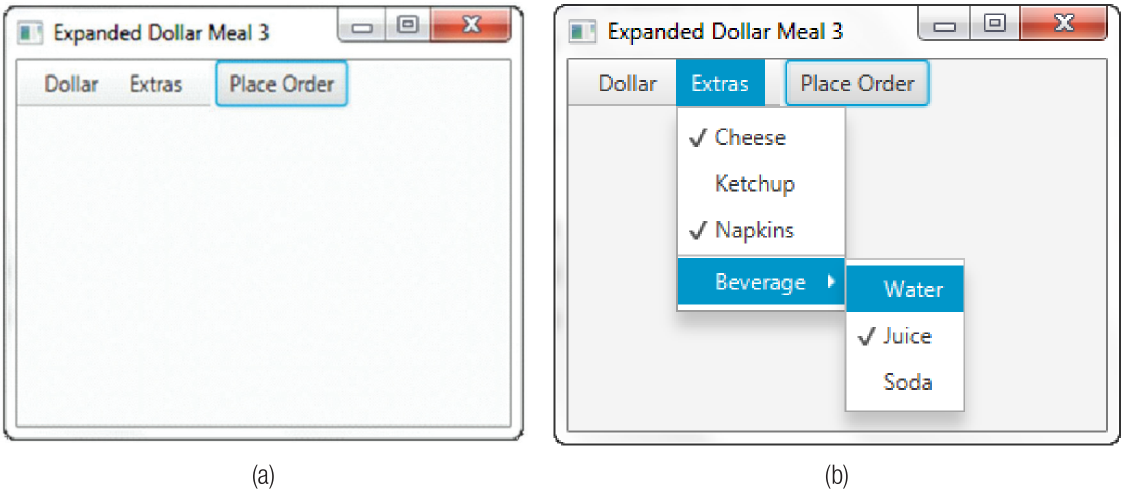


Figure 12.12
Check Box and Radio Box items in a drop-down menu.

When the user selects a check menu item or a radio menu item displayed on a menu, an `ActionEevent` occurs. The code to execute the method `extrasMenuItemHandler` when the click occurs on the check menu item `cheeseItem` is given below. As previously discussed, its signature would be `extrasMenuItemHandler(ActionEvent e)`and its code would invoke the `getSource()` method on the argument passed to it to determine which menu item was selected.

```
cheeseItem.setOnAction(e-> extrasMenuItemHandler(e));
```

Normally, on a menu that permits multiple selections, we would not want to perform the application-dependent processing until user selections were complete and a button was clicked. The code to add the `Button` object `placeOrder` to the menu bar area of the window shown in

Figure 12.12 whose container is named `pane`, and execute the method `placeOrderItemHandler` when the button is clicked, is given below.

```
Button placeOrder = new Button("Place Order");
placeOrder.setLayoutX(134);
placeOrder.setLayoutY(1);
placeOrder.setOnAction(e-> placeOrderItemHandler(e));
pane.getChildren().add(placeOrder);
```

12.3.2 Pop-Up Menus

A pop-up menu is a space-saving alternative to a menu bar based, drop-down menu. Unlike drop-down menus, pop-up menus are associated with a particular component in a graphical interface, and they remain invisible until the user performs a platform-dependent action on the GUI component. A common action on the component is a right mouse click.

Pop-up menus are instances of the class `ContextMenu` and can be created using the class's default constructor:

```
//Create a pop-up menu object
ContextMenu aPopUpMenu = new ContextMenu();
```

The techniques discussed in the previous section used to add menu items, submenus, and separators to drop-down menus, and to perform application-dependent processing when a menu item is selected, are the same techniques used for pop-up menus. The only difference is that a pop-up menu is not added to a menu bar. Instead it is associated with a GUI component. When that GUI component is right clicked, the pop-up menu becomes visible (pops up) with its upper left corner positioned at the mouse pointer.

To associate a pop-up menu with a component, the `setContextMenu` method is invoked on the GUI component, and the pop-up menu object is passed to the method. When the user performs the platform-dependent action on the component (e.g., right clicking the component), the menu becomes visible. The last line of the following code sequence associates the `ContextMenu` object `aPopUpMenu` with the `Label` component `aLabel`.

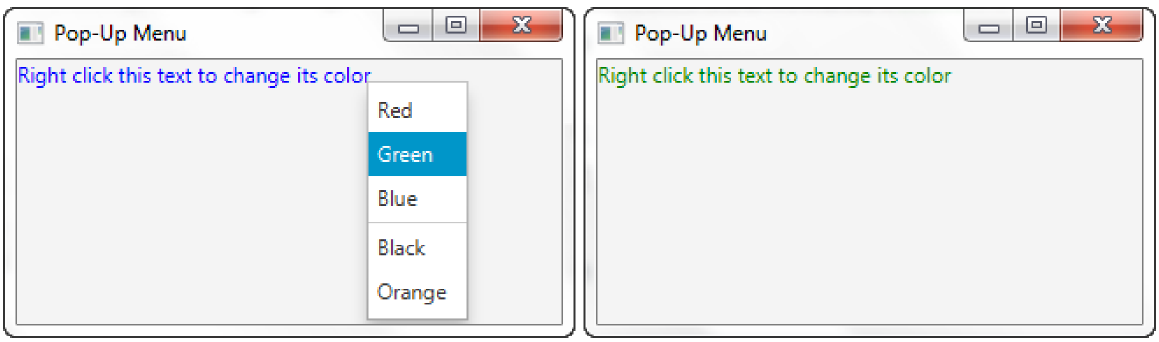
```
//Associating a pop-up menu with a component
aLabel = new Label("Right click this text to change its color");
aLabel.setTextFill(Color.BLUE);
aLabel.setContextMenu(aPopUpMenu);
```

The application `PopUpMenu` shown in Figure 12.13 creates the pop-up menu shown in Figure 12.14a, which appears when the text displayed in the window is right clicked. Figure 12.14b shows the window after the user selected Green from the menu by left clicking it. This application uses a coding technique similar to that used in Figure 12.10. The highlighted lines in Figure 12.13 identify the adaptation of that technique to this application, and also identify the lines that change the color of text displayed in the window to green.

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class PopUpMenu extends Application
{
   Label aLabel;
   MenuItem red, green, blue, black, orange;

   @Override
   public void start(Stage primaryStage)
   {
      // Create the pop-up menu
      ContextMenu popUpMenu = new ContextMenu();

      // Create the menu items and event handler
      red = new MenuItem("Red");
      green = new MenuItem("Green");
      blue = new MenuItem("Blue");
      black = new MenuItem("Black");
      orange = new MenuItem("Orange");
      red.setOnAction(e-> colorChangeHandler(e));
      green.setOnAction(e-> colorChangeHandler(e));
      blue.setOnAction(e-> colorChangeHandler(e));
      black.setOnAction(e-> colorChangeHandler(e));
      orange.setOnAction(e-> colorChangeHandler(e));
      SeparatorMenuItem separator1= new SeparatorMenuItem();

      // Add the menu items to the menu
      popUpMenu.getItems().addAll(red, green, blue, separator1,
                                  black, orange);

      // Create a component and associate it with the menu
      aLabel = new Label("Right click this text to change its color");
      aLabel.setTextFill(Color.BLUE);
      aLabel.setContextMenu(popUpMenu);

      StackPane root = new StackPane();
      root.getChildren().add(aLabel);

      Scene scene = new Scene(root, 300, 150);

      primaryStage.setTitle("Pop-Up Menu");
      primaryStage.setScene(scene);
      primaryStage.show();
```

```
    }

    public void colorChangeHandler(ActionEvent e)
    {
        if(e.getSource() == red)
        {
            aLabel.setTextFill(Color.RED);
        }
        else if(e.getSource() == green)
        {
            aLabel.setTextFill(Color.GREEN);
        }
        else if(e.getSource() == blue)
        {
            aLabel.setTextFill(Color.BLUE);
        }
        else if(e.getSource() == black)
        {
            aLabel.setTextFill(Color.BLACK);
        }
        else if(e.getSource() == orange)
        {
            aLabel.setTextFill(Color.ORANGE);
        }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 12.13
The application `PopUpMenu`.



(a) (b)

Figure 12.14
The Pop-up menu produced by the application `PopUpMenu` used to change the text's color to green.

12.4 FILE-CHOOSER AND COLOR-PICKER DIALOG BOXES

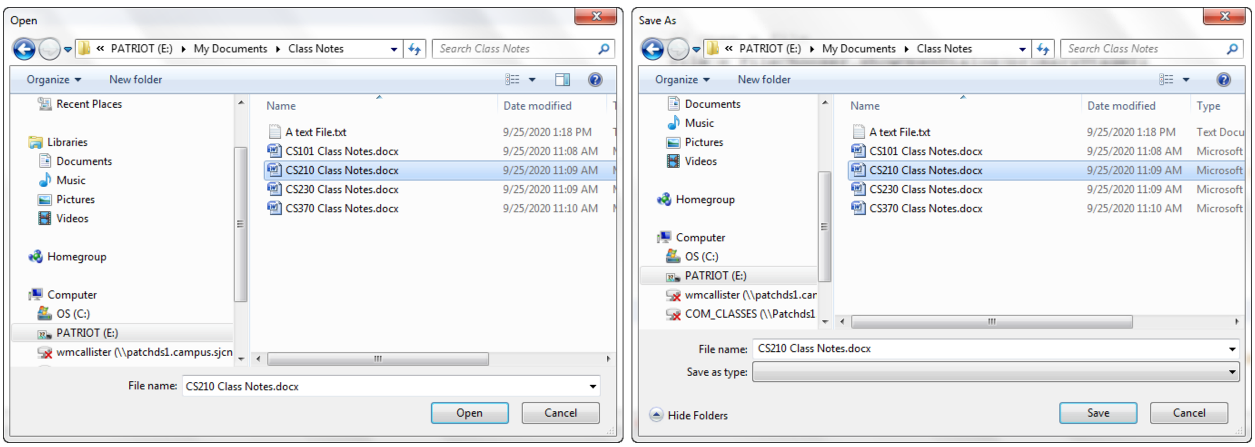
FX provides three dialog boxes that can be used to facilitate commonly performed user tasks: specifying the path to a file to be opened, specifying the path to a file to be saved, and specifying a color to be used in a graphics application. The `FileChooser` class contains methods that display file-open and file-save dialog boxes. An object in the `ColorPicker` class is a dialog box containing a predefined palette of colors from which to choose, and provides the ability to define a custom color.

12.4.1 File-Chooser Dialog Boxes

Figure 12.15a shows an Open file-chooser dialog box, and Figure 12.15b shows a Save file-chooser dialog box. Both dialog boxes are displayed by the application `FileChoosers`, shown in Figure 12.16.

Line 21 displays the Open file-chooser dialog box by invoking the `showOpenDialog` method on the `FileChooser` object `fileChooser` declared on line 17, passing the method the application's stage object. Line 21 of the application does not complete its execution until the user clicks the dialog box's Open or Cancel buttons or closes the dialog box. Until one of these events occurs, the user can browse the system's file structure to locate and select a file to be opened, or the user can click a folder's name and then type the name of a file name into the File Name text field.

When a folder is selected by double-clicking its name, the path to the folder appears in the text field at the top of the dialog box, and its subfolders are displayed. When a file is selected by clicking its name, the file name is displayed in the dialog box's File Name text field. A subsequent click of the Open button will close the dialog box. Whenever the Cancel button or the X at the top right of the dialog box is clicked, the dialog box closes.



(a) (b)

Figure 12.15
File-chooser Open and Save dialog boxes.

When the dialog box closes, the method `showOpenDialog` returns the address of a `File` object, or `null` if the dialog box was closed by clicking the Cancel button or the X. When that is not the case, the Boolean condition of the `if` statement on line 22 is `true`, and the returned `File` object contains a string that represents the path to the file. To fetch this string, the `File` class's `getPath` method is invoked on the returned object. Lines 24–25 use this method to fetch and output the selected file's path to the system console.

To display the file when the user closes the dialog box by clicking Open, the `open` method is invoked on a `Desktop` object and the method is passed the file path. The following two lines would be added to the code block of the `if` statement on line 22 to display the file to the user.

```
Desktop desktop = Desktop.getDesktop();
desktop.open(file);
```

Lines 29–34 use a sequence of code similar to lines 21–26 to display a Save file-chooser dialog box and output the path to the saved file to the system console. The only difference is that line 29 invokes the `showSaveDialog` method instead of the `showOpenDialog` method.

The output produced by the application is shown in Figure 12.17. To generate this output, the user clicked the Open button of dialog box shown in Figure 12.15a, and then clicked the Save button of the dialog box shown in Figure 12.15b.

NOTE *The* look and feel *of the dialog boxes displayed in Figure 12.15 is platform dependent.*

```
1   import java.io.File;
2   import java.io.IOException;
3   import javafx.application.Application;
4   import javafx.event.ActionEvent;
5   import javafx.scene.Scene;
6   import javafx.scene.layout.Pane;
7   import javafx.stage.FileChooser;
8   import javafx.stage.Stage;
9
10  public class FileChoosers extends Application
11  {
12     @Override
13     public void start(Stage primaryStage) throws IOException
14     {
15        Pane root = new Pane();
16
17        FileChooser fileChooser = new FileChooser();
18        File file;
19
20        // Open a file
21        file = fileChooser.showOpenDialog(primaryStage);
22        if (file != null) // Open was clicked
```

```
      {
         System.out.println("The path to read the file from is:\n" +
                             file.getPath());
      }

      // Save a file
      file = fileChooser.showSaveDialog(primaryStage);
      if (file != null) // Save was clicked
      {
         System.out.println("The path to write the file to is:\n" +
                             file.getPath());
      }

      Scene scene = new Scene(root, 400, 250);
      primaryStage.setTitle("File Chooser Dialogs");
      primaryStage.setScene(scene);
      primaryStage.show();
   }

   public static void main(String[] args)
   {
      launch(args);
   }
}
```

Figure 12.16
The application `FileChoosers`.

Console output:
The path to read the file from is:
E:\My Documents\Class Notes\CS210 Class Notes.docx
The path to write the file to is:
E:\My Documents\Class Notes\CS210 Class Notes.docx

Figure 12.17
The output produced by the application `FileChoosers`.

12.4.2 Color-Picker Control

A `ColorPicker` object is a control component that can be added to a pane container. It is used in applications where the user will be asked to select a color. The selected color can then be used as the color of another component or a drawn shape. To facilitate this process, the color-picker contains a pallet of 120 predefined colors that the user can select a color from, and a drop-down window that can be used to compose a custom color. The color pallet object also contains a color-chooser that displays the current color. Initially it is the only part of the color-picker visible.

The three parts of a color-picker component are shown in Figure 12.18. When the color-chooser is clicked, the pallet is made visible. When the text Custom Color displayed at the bottom of the pallet is clicked, the drop-down custom color window is also made visible.

The `ColorPicker` class contains a default constructor and a one-parameter constructor that can be passed an instance of the class `Color`. Most often it is passed one of the `Color` class's static constants to specify the current color. The below code sequence declares the `ColorPicker` object `aColorPicker`, with a current color of orange, that is displayed in Figure 12.18.

```
ColorPicker aColorPicker= new ColorPicker(Color.ORANGE);
```

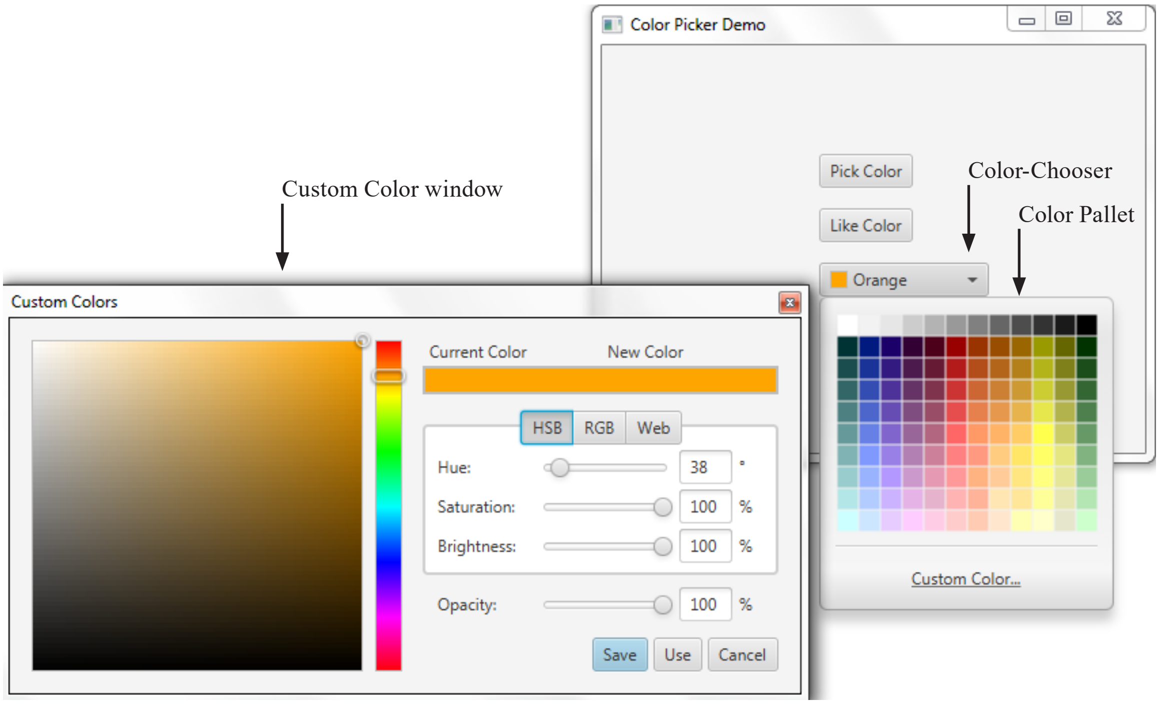


Figure 12.18
The three parts of a `ColorPicker` object.

Displaying Color-Picker

Like other control components a color-picker's visible property is `true` by default, and so it is displayed when the container it is added to is displayed. To prevent it from being displayed, its visible property is set to `false` by invoking the `setVisible` method on it and passing the method the Boolean value `false`. At the point in the program's execution when the color-chooser should be displayed, `setVisible` would be invoked on it and passed the value `true`.

Alternately, the color-picker can be added to a visible container at the point in the program's execution when the color-chooser should be displayed, and removed from the container when it should not be displayed.

Color-Picker Events

When the user clicks any of the predefined colors on the color pallet, or the Save or Use buttons in the custom color window, an `ActionEvent` occurs and the current color becomes the color clicked in the color pallet or the new color composed in the custom color window. The technique used to register the event handler associated with these actions on a color-picker is the same technique used to register a click on a button. For example, the below line of code would cause the method `handle` to be invoked and passed an `ActionEvent` object when these click actions occurred on the color-picker object `aColorPicker`.

```
aColorPicker.setOnAction(e-> handle(e));
```

The method `handle` shown below would be coded inside the application. It would perform the application-dependent processing in response to the color-picker event. If the application-dependent processing of other components' action events were also coded inside this method, the `getSource` method would be invoked on the object passed to the method to determine on which component the event occurred. The code block of the `if` statement inside the method `handle` shown below would execute when the event occurred on the color-picker object `aColorPicker`.

```
public void handle(ActionEvent e)
{
   if(e.getSource() == aColorPicker)
   {
      System.out.println("New current color selected");
   }
   //some other action event occurred
   else if…
}
```

Fetching and Applying the Selected Color

A color-picker object contains a data member named `value` that stores the object's current color property. Initially, `value` is assigned the color passed to the color-picker's constructor. Subsequently it is assigned the color selected by the user, or the custom color created by the user, or the color passed to the color-picker's method `setValue`. To fetch the current color, the class's `get–Value` method is invoked on the color-picker object. The method returns an instance of the class `Color`.

If lines 15 and 21 of the application presented in Figure 11.24 were replaced with the lines 1–3 shown below, and the method handle (lines 5–11 shown below) was added to the application, then the user would be able to select the snowman's hat color from a color-picker object before the snowman was displayed. The variables `aColorPicker` and `root` on lines 1 and 3 below would have to be declared at the class level in Figure 11.24 so that they could be accessed within the below code.

```
1  aColorPicker = new ColorPicker();
2  aColorPicker.setOnAction(e -> handle(e));
3  root.getChildren().add(aColorPicker);
4
5    public void handle(ActionEvent e)
```

```
{
   Color c = aColorPicker.getValue();
   sm1 = new SnowManOnACanvas(280, 40, c);
   root.getChildren().remove(aColorPicker);
   root.getChildren().addAll(sm1.getCanvas()); //add(canvas)
}
```

To change the color of a `Button` object or `Pane` object such as a `VBox` pane, the method `setStyle` is invoked on the object and passed a CSS string that describes its background color. A detailed description of FX's CSS feature is beyond the scope of this book. However, constructing the CSS string from the `Color` object returned from the `getValue` method is a simple feature of CSS to illustrate.

The below four lines of code changes the background color of the `VBox` pane named `root`, declared on line 1, to the current color of the color-picker object `aColorPicker`.

```
1   VBox root = new VBox();
2   String c = aColorPicker.getValue().toString();
3   c = "#" + c.substring(2,8);
4   root.setStyle("-fx-background-color: " + c);
```

Line 2 invokes the `Color` class's `toString` method on the `Color` object returned from the `getValue` method, and stores the address of the returned string in the variable `c`. Then line 3 gleans character 2 through 7 from that string. These six characters are the concatenation of the two digit hexadecimal intensities of the colors red, green, and blue that make up the selected color. For example, `ff0000` would be the brightest color of red possible. This string preceded by the three characters `"#"` is then concatenated into the end of a CSS property name on line 4 and passed to the `setStyle` method invoked on the VBox `root`.

Figure 12.19 contains the code of the application `ColorPickerDemo`. When launched it displays the window shown in Figure 12.20a. The user can click the Pick Color button to change the background color of the window to a different color selected from a color-picker. The process can be repeated until the user is satisfied with the window's color, then a click of the Like Color button removes the two buttons from the window (Figure 12.20c).

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ColorPickerDemo extends Application
{
   ColorPicker aColorPicker= new ColorPicker(Color.ORANGE);
   Button pickColor, likeColor;
   VBox root;

   @Override
```

```
    public void start(Stage primaryStage)
    {
        root = new VBox();
        pickColor = new Button("Pick Color");
        likeColor = new Button("Like Color");

        aColorPicker.setOnAction(e-> handle(e));
        pickColor.setOnAction(e-> handle(e));
        likeColor.setOnAction(e-> handle(e));

        root.setAlignment(Pos.CENTER);
        root.setSpacing(15.0);
        root.getChildren().addAll(pickColor, likeColor);

        Scene scene = new Scene(root, 400, 300);

        primaryStage.setTitle("Color Picker Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void handle(ActionEvent e)
    {
        if(e.getSource() == pickColor &&
           !root.getChildren().contains(aColorPicker))
        {
            root.getChildren().add(aColorPicker);
        }
        else if(e.getSource() == aColorPicker)
        {
            String c = aColorPicker.getValue().toString();
            c = "#" + c.substring(2,8);
            root.setStyle("-fx-background-color: " + c);
            root.getChildren().remove(aColorPicker);
        }
        else if(e.getSource() == likeColor &&
                !root.getChildren().contains(aColorPicker))
        {
            root.getChildren().removeAll(pickColor, likeColor);
        }
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

Figure 12.19

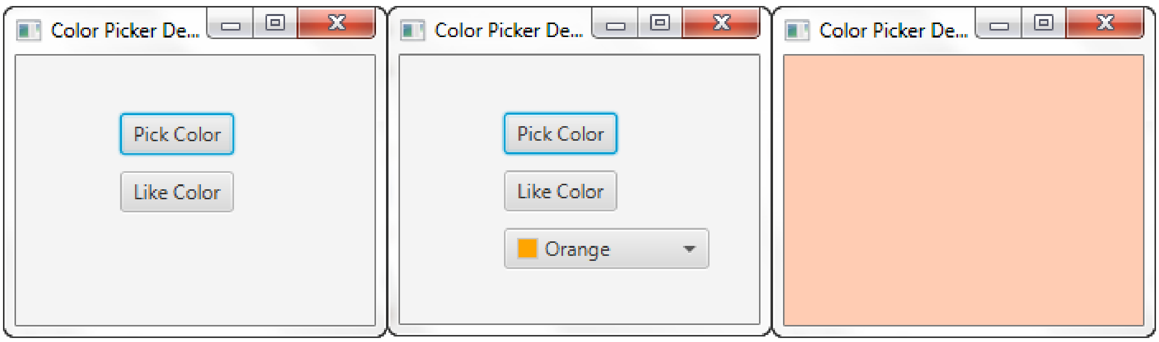
The application `ColorPickerDemo`.

Line 11 declares the color-picker object `aColorPicker` whose current color is orange, and lines 19–20 declare the two buttons. Lines 22–24 associates their `ActionEvent` handlers with the same method, `handle`, coded on lines 37–56. Line 28 adds the two buttons to a `VBox` layout pane named `root` declared in line 18. The VBox is added to application's scene on line 30.

When the Pick Color button (named `pickColor`) is clicked *and* the color-picker object (named `aColorPicker`) is not contained in the VBox, the Boolean condition of the `if` statement on line 39 is `true`. Line 42 adds the color-picker to the VBox, which causes its color-chooser to be displayed as shown in Figure 12.20b. A subsequent click on the color-chooser displays the color pallet, but does not generate an action event.

When a color is selected by clicking one of the color pallet's colors or selected from the custom color window by clicking its Save or Use buttons, an action event occurs on the color-picker object `aColorPicker`. Since the color-picker is displayed, the `if` statement on line 44 of the event handler method executes and its Boolean condition is `true`. Lines 46–47 in the code block of this `if` statement fetches the selected color, and then line 48 makes that the color of the VBox pane. Line 49 removes the color-picker object from the VBox returning the window to the one displayed in Figure 12.20a, but with its background color changed to the selected color, e.g., orange.

If the user clicks the Like Color button when the color-picker is not displayed, the `if` statement on line 51 executes and its Boolean condition is `true`. Lines 54 of this `if` statement's code block removes the two buttons from the VBox, and the applications window is the one displayed in Figure 12.20c.



(a) (b) (c)

Figure 12.20
The progression of the application `ColorPickerDemo`'s window as the user interact with it.

12.5 LAMBDA EXPRESSIONS AND FUNCTIONAL INTERFACES: A SECOND LOOK

Lambda expressions were added to Java in Version 8. Generally speaking, they can be thought of as a concise implementation of an abstract method whose signature is defined in an interface that contains one, and only one, abstract method. This type of interface is referred to as a *functional* interface. It is also called a Single Abstract Method (SAM) interface. The interface can also contain static constants, and static and default method implementations. The abstract method can be a void or non-void method, and its signature can contain parameters.

As we have previously mentioned, the interface `EventHandler` contains only one abstract method, `handle`, and so it is a functional interface. Therefore, in Chapters 11 and 12 we could implement handle's code block with a Lambda expression.

From the translator's viewpoint, the name of the interface that contains the abstract method's signature being implemented by the Lambda expression is implied from the context in which the Lambda expression is coded. For example, the Lambda expression on line 22 of Figure 12.19 is coded in the position where the argument passed to `setOnAction's` one parameter would normally be coded. Since this parameter's type is the functional interface `EventHandler`, the Lambda expression is considered to be an implementation of that interface's abstract method `handle`.

Syntax of a Lambda Expression

To understand the execution path of a Lambda expression we need to become failure with its syntax. The syntax of a Lambda expression is:

Parameter list -> Code Body

The two keystrokes symbol -> is called the "arrow token" or the "lambda operator," and it is always coded between the parameter list and the code body. Like the two keystrokes of the logic equals token ==, the two keystrokes of the arrow token must be sequential, i.e., no white space is typed in between them.

The number of parameters in the parameter list must be the same as the number of parameters in the signature of the abstract method the Lambda expression is implementing. The syntax of the parameter list is the same syntax used in non-Lambda expression method implementations except:

- The parameter types can usually be omitted e.g., `(x, y, z) ->`. The translator will infer them from the abstract method's signature.
- For one parameter the parentheses can be omitted: e.g., `x ->.`
- When the parameter list is empty we just code the parentheses: `( ) ->`.

The names of the parameters in the lambda expression need not be the same as the names of the parameters in the abstract method's signature. As in named (non-Lambda) method implementations, the parameters are value type parameters.

The Code Body of a Lambda expression cannot return a value if the method it is implementing is a void method, and it must return a value of the type specified in the signature of a non-void method it is implementing.

The Code Body of the Lambda expression uses the same syntax as a code body in a named method except if the code body consists of just *one* statement, then:

- The open and close braces { } that begin and end the Code Body need not be coded. If they are not coded a semicolon is not included at the end of the statement.

- If the Lambda expression is implementing a non-void method, the returned value would be the result of the one statement. For example, the Lambda expression `x -> x + 1` returns the value passed to the method incremented by one.
- If the Lambda expression is implementing a non-void method and the one statement is inside a code block, then the keyword `return` must be used. For example: `x -> {return x + 1;}`.

The Lambda expression coded on line 22 of Figure 12.19, `e -> handle(e)`, conforms to these syntax rules. It is an implementation of the one-parameter method `handle` defined in the function interface `EventHandler`. Its one statement simply invokes a method with the same name coded within the application, and passes that method the argument passed it by the runtime environment. As a result, the Lambda expression begins with one parameter that, in this case, is not enclosed in parentheses. In addition, it contains only one statement, making the open and close braces optional. Since they were not used, a semicolon was not coded at the end of the statement.

When a Lambda expression is implementing a non-void method, the type of the item returned from the Lambda expression must be compatible with the return type coded in the abstract method's signature.

The following are all valid lambda expressions.

```
//Implements a void method with no parameters
 ( ) ->   System.out.println("Output from a Lambda Expression.") or
 ( ) -> { System.out.println("Output from a Lambda Expression."); }

//Implements a void method with two parameters using two statements
 (x, y) -> { System.out.println("x = " + x + " y = " + y);
             System.out.println("x + y = " + (x + y));
           }

//Implements a non-void method with two parameters
//that returns 2*x + 3*y
 (x, y) -> { x = x * 2;
             y = y * 3;
             return (x + y);
           }
```

Lambda expressions can also contain decision statements, loops, and method invocations.

Executing a Lambda Expression with an Abstract Method Invocation

To execute in the code of a Lambda expression, the Lambda expression is assigned to a variable whose type is the name of the interface. There are three coding contexts we can use to do this. Then that variable is used to invoke the abstract method.

The most overt of these three coding contexts is shown in Figure 12.21. The lambda expression coded on lines 4–6 is assigned on line 4 to the variable `b`, whose type is the name of the functional interface the Lambda expression is implementing. This functional interface, named `AFuncIterface`, is coded on lines 12–15. Then that variable is used on line 8 to invoke the abstract method `twoAsAndThreeBs` defined in the functional interface. This causes the Lambda expression's code previously assigned to the variable `b`, to become the code of the interface's abstract method, and the code executes. Since the method is passed 1 and 2 on line 8, the integer 8 is returned from the method and then output on line 9.

```
1    public static void main(String[] args)
2    {
3       //assign the Lambda expression to the AFuncInterface variable b
4       AFuncInterface b = (x, y) -> { x = 2 * x;
5                                      y = 3 * y;
6                                      return x + y; };
7       //invoke the abstract method on the AFuncInterface variable b
8       int ans = b.twoAsAndThreeBs(1, 2);
9       System.out.println(ans); //outputs 8
10   }
11
12   public interface AFuncInterface //the functional interface
13   {
14      int twoAsAndThreeBs(int a, int b); //the abstract method
15   }
```

Figure 12.21
Assigning a Lambda expression to a variable.

A second coding context used to assign a Lambda expression to a variable whose type is the name of the interface is shown in Figure 12.22. On line 4 the method `output2AsAnd3Bs`, coded on lines 9–14, is invoked and the Lambda expression is passed to its parameter. This parameter's type is the name of the functional interface containing the abstract method that the Lambda expression is intended to implement, just as the variable `b`'s type was in previously discussed context.

As is the case with any argument passed to a parameter of a method, before the method begins execution the argument (in this case the Lambda expression) is *assigned* to the parameter (in this case the parameter named `e`). Then that variable is used on line 12 to invoke the abstract method `twoAsAndThreeBs` defined in the functional interface (lines 16–19). This causes the Lambda expression's code previously assigned to the parameter `e`, to become the code of the interface's abstract method, and the code executes. Since the method is passed 1 and 2 on line 12, the integer 8 is returned from the method and then output on line 13.

```
1    public static void main(String[] args)
2    {
3       //pass the Lambda expression to a method's AFuncInterface parameter
4       output2AsAnd3Bs( (x, y)-> { x = 2 * x;
5                                   y = 3 * y;
6                                   return x + y;} );
7    }
8
9    public static void output2AsAnd3Bs(AFuncInterface e)
10   {
11      //invoke the abstract method using the parameter e
12      int ans = e.twoAsAndThreeBs(1, 2);
```

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

public interface AFuncInterface //the functional interface
{
    int twoAsAndThreeBs(int a, int b); //the abstract method
}
```

Figure 12.22
Assigning a Lambda expression to the parameter of a void method.

The coding context of Lambda expressions shown in Figure 12.22 offers us insights into the event handler registration process used in Chapters 11 and 12 where a Lambda expression was passed to the parameter of non-static methods that began with the prefix `setOn`. After receiving the Lambda expression, these methods save the parameter passed to it and then use it to invoke the method `handle` when the run-time event occurs on the object that invoked it.

This coding context is also used to pass Lambda expressions to methods in the `IntStream` class that is part of the `stream` package. The `stream` package was added to Version 8 of Java to incorporate the functional programming paradigm into it.

The third coding context used to assign a Lambda expression to a variable whose type is the name of the functional interface is shown in Figure 12.23. Here the variable is assigned the item returned from an invocation of a non-void method named `lambdaBuilder` (line 4). The method (lines 10–13) is passed the Lambda expression and simply returns the argument it receives. Then the returned value is passed to a method (line 7) that uses it to invoke the abstract method (line 18). This causes the lambda expression's code to become the code of this execution of the abstract method.

```
1   public static void main(String[] args)
2   {
3      // pass the lambda expression to a non-void method
4      AFuncInterface e = lambdaBuilder( (x, y)-> { x = 2 * x;
5                                                   y = 3 * y;
6                                                   return x + y;} );
7      output2AsAnd3Bs(e);
8   }
9
10  public static AFuncInterface lamdaBuilder(AFuncInterface e)
11  {
12     return e;
13  }
14
15  public static void output2AsAnd3Bs(AFuncInterface e)
16  {
17     // invoke the abstract method using the parameter e
18     int ans = e.twoAsAndThreeBs(1, 2);
19     System.out.println(ans);
20  }
```

```java

public interface AFuncInterface // the functional interface
{
   int twoAsAndThreeBs(int a, int b);
}
```

Figure 12.23
Assigning a Lambda expression to the parameter of a non-void method.

12.6 SCENE BUILDER

Scene Builder is a Java FX programming environment created by Oracle to facilitate the creation of graphical user interfaces within an FX application. It provides the same type of tools that Microsoft's Visual Studio provides for quickly developing user interfaces within Visual Basic and Visual C++ programs.

The programmer is presented with a window and a set of drop-down menus that display lists of controls, containers, menus, and shapes that can be selected for inclusion in an application's GUI. The selected item is then dragged to its position within the displayed window. The item can be resized by dragging its edges, and its text property can be set by simply clicking the item and typing on it.

Other properties, such as color, font, spacing, and alignment can be easily set by clicking the component and selecting the desired property from a drop-down menu of component's properties. As this process is being used to develop the user interface, the environment is generating the corresponding code.

The bottom line is that the user interface can be rapidly created without the programmer writing the code to generate it. In addition, the application-dependent processing is simply written into the event handler methods created and registered by the environment.

Scene Builder is incorporated into the popular IDEs such as NetBeans and Eclipse, and Oracle provides a set of online documentation describing its use. NetBeans also contains a drag-and-drop environment for creating GUI's using Swing components.

12.7 CHAPTER SUMMARY

The GUI components check box, radio button, combo box, and list view are used to select one or more inputs from a set of valid inputs. These components are instances of the classes `CheckBox`, `RadioButton`, `ComboBox`, and `ListView`, respectively. Ordinarily, either a set of radio buttons or a combo box is used when the choices are mutually exclusive, and a set of check boxes or a list view is used when this is not the case. To make radio buttons mutually exclusive, their `ToggleGroup` property is set to the same `ToggleGroup` by invoking the `setToggleGroup` method on them.

When the set of input choices is large, a combo box or a list view is preferred over radio buttons and check boxes because they both can display a scroll bar. This permits the user to view a large number of selection choices within a small space on the program's window. The number of selections displayed in a combo box's drop-down panel can be specified by invoking the `setVisibleRowCount` method on the combo box object and passing it an integer row count.

The annotation to be displayed next to a radio button or a check box can be passed to the constructor invoked to create these objects. The selections displayed in a combo box or a list view is added to them by passing their constructor an instance of an `ObservableList` containing the selections, or by invoking the `add` or `addAll` methods on the observable list whose address is stored in combo box's or list view's `items` data member. The selections are passed to these methods as arguments.

Multiple non-sequential values in a list view can be selected by clicking them while holding down the Ctrl key on the keyboard. Multiple sequential values can be selected by clicking the first value in the sequence, holding down the Shift key, and then clicking the last value in the sequence. The ability to select one or more values from a list is not its default mode, however the multiple selection mode can be specified.

Normally, a set of check boxes, a set of radio buttons, a combo box, or a list view is added to an instance of a Pane, and the pane is then added to the window's scene or another pane. This makes the components easier to position in the window, and the pane's border can be made visible to give the impression that the boxes or buttons it contains are part of a set.

In most applications that use check boxes, radio buttons, combo boxes or list views, the interface contains a button that is clicked after the user finalizes the selections and the application-dependent processing is performed within the button's event handler. When it is important to perform processing immediately after a selection is made, an event handler is associated with the check box, radio button, combo box, or list view. The selections produce `ActionEvents`, and so the `setOnAction` method is invoked on the component to register its event handler.

When the application-dependent processing on several components is processed by the same event handler, the `getSource` method can be invoked on the argument passed to the event handler to determine which component caused the event. To determine if a particular check box or radio button has been selected, the `isSelected` method is invoked on the object. This method returns `true` or `false`. The selection made in a combo box or all the selections made in a list view can be determined using the methods presented in Table 12.2, which also describes a method to display a text field in a combo box into which the user can type a non-displayed selection.

The Java API supports two types of menus, drop-down menus and pop-up menus, which are used to construct a user-friendly interface that presents a group of valid input selections within a relatively small portion of the program's window. The annotation associated with these components is passed to their class's one-parameter constructor when they are created.

Drop-down menus are added to, and displayed on, a menu bar whose location in the window is platform dependent. Pop-up menus are associated with other GUI components and remain invisible until the user performs a platform-dependent action (e.g., a right mouse click) on the associated component. The `addAll` method is invoked on a menu bar object to add items and drop-down menus to it, and invoked on a drop-down or pop-up menu object to add a menu selection item or a drop-down (sub) menu to them. Horizontal separator lines can also be added to these menus to visually group their related elements.

Menu selection items are instances of the classes `MenuItem`, or `CheckMenuItem`, or `RadioMenuItem` that are displayed as the string passed to their constructor. Check menu and radio menu items provide the functionality of check boxes and radio buttons, and a check box is displayed to the left of their text.

When the user selects a menu item, an `ActionEvent` occurs. The event is serviced using the same techniques used to service action events on `Button` objects, and the `getSource` method is invoked on the object passed to the event handler method to determine which menu selection item was selected. When the menu selection items are check boxes, processing is usually performed when a button is clicked after the user finalizes the selections, and the `isSelected` method is invoked within the event handler method on the menu selection items to determine the selections.

In addition to these GUI components, the `FileChooser` class provides methods to facilitate the commonly performed user tasks of opening and saving files. Dialog boxes to perform these tasks are displayed by creating an instance of this class and invoking the class's `showOpenDialog` or `showSaveDialog` method on the object. Both methods return a `File` object that contains the selected file path, which can be fetched by invoking the `getPath` method on the returned object.

An instance of the class `ColorPicker` provides a palette of colors from which to choose a color, and also provides the ability to define a custom color. When the `ColorPicker` object is added to a visible container, it is displayed. When a color is selected, an `ActionEvent` occurs. The selected color can be determined by invoking the `getValue` method on the `ColorPicker` object. The two character hexadecimal intensities of the selected color's red, green, and blue intensities are characters 2 through 7 of the string version of the object returned from the `getValue` method.

Knowledge Exercises

1. True or false:

a) Radio buttons are commonly used to select one or more inputs from a set of valid inputs.
b) Adding check boxes or radio buttons to a pane makes them easier to position.
c) Combo boxes are used to make multiple selections from a set of valid inputs.
d) Combo boxes or lists views are used when the number of input choices is large.
e) A scroll bar can be displayed in a combo box.
f) A scroll bar is not displayed in a list view if all of the items can be displayed.
g) Multiple values can be selected from a list view component at one time.
h) Multiple items can be selected from a combo box at one time.
i) The elements displayed in a list and/or combo box are defined as an array of objects.
j) GUI components can be made invisible.
k) Combo boxes can be edited, allowing a user to type a choice into a text field.
l) List views can be edited, allowing a user to type a choice into a text field.

m) Java supports drop-down but not pop-up menus.

n) Radio buttons and check boxes can be added to a menu.

2. Give examples of when you would use a group of check boxes and when you would use a group of radio buttons.
3. State when you would use a combo box and when you would use a list view.
4. Explain how or give examples:

   a) Radio buttons can be grouped together to make them mutually exclusive.

   b) A border can be displayed around a `Pane` object.

   c) A border's color can be changed from its default color.
5. Compare and contrast the features of combo boxes and lists.
6. Discuss the differences between drop-down and pop-up menus.
7. State an advantage of using menus in your applications.
8. Briefly explain the function of the `FileChooser` dialog boxes.
9. What is returned from the method invoked to display a `FileChooser` dialog box?
10. Briefly explain the function of a `ColorPicker` object.
11. Give a line of code to specify that the method `saladClickHandler`() should be invoked when the user clicks the check box object `salad`.
12. Give the code to add the three items of your choice to the list view `snacks`.
13. Give the code to determine if the check box `soda` was selected.
14. Give the class of each of the following components:

    a) Check box **b)** Radio button

    c) Combo box **d)** List view

    e) Drop-down menu **f)** Pop-up menu

Programming Exercises

1. Design, write, and test a GUI application for the Speedy Cable Service. Include a menu to offer the user any combination of the following service options: basic, movie, sports, premium, and learning. When a user clicks the Calculate button, the monthly bill for all of the services selected should be computed and output in a dialog box. The monthly costs are as follows: basic is $30, movie is $15, sports is $20, premium is $30, and learning is $12. Customers can also select high or extra-high definition for a fee of $10 or $15 per month respectively.
2. Using a GUI design, write and test an application for a bank that offers the user the following choices: make a deposit, make a withdrawal, or check the balance of an account from a drop-down menu. Dialog boxes should be used for user input and output after a menu item is selected.

3. Colorful Sports Inc. just hired you to write a pop-up menu GUI application that their customers will use to order winter clothing from a selection of three custom-colored items they sell: shirts for $30, parkas for $150, and gloves for $15. The customer can select any or all of the items and can specify the size (small, medium, large, or extra large) for each item selected. When an item is selected, present the users with a color-chooser from which they can select the color of the item. Provide a Place Order button that calculates and outputs the total cost to the GUI, including an 8% sales tax, the items ordered, and a swatch of the color of each item to the GUI. Include a Reset button on the GUI that clears the output and all of the selections that were made. Design the GUI.
4. Write the code and test the application described in Exercise 3.
5. Design a GUI for Sam's Sandwich Shop to allow users to place orders for sandwiches. The selections should include (but are not limited to) the following items: the choice of bread (Italian, wheat, rye), one or more fillings (ham, cheese, turkey, tuna, lettuce, tomato, mayonnaise, and mustard), and one or more beverages (soda, water, and coffee). After the selections are made, allow the user to click a button and view the order in a dialog box. Provide a Place Order button, and Reset button that clears the output and all of the selections that were made.
6. Write a program to implement the design for Sam's Sandwich Shop in Exercise 5 and output the order.
7. Design a GUI for the Tanya's Tour Trips travel agency that her customers can use to select the year, month, and day of their trip, the number of people traveling (up to four people), and a group of cities to be visited from a list of 30 cities. When the Book It button is clicked, the date of travel (mm/dd/yy) is output to the GUI along with a scrollable list of the cities to be visited. The GUI should also provide a Reset button that clears the output and all of the selections.
8. Write a program to implement the design for Tanya's Tour Trips in Exercise 7.
9. Write a GUI application that asks the user to select three colors from a set of predefined colors or a user defined custom color. After the selections are made, three rectangular swatches of the colors should be displayed in the window.

Enrichment

Investigate some of the features of Cascading Style Sheets (CSS). Here are some suggested resources:

www.w3schools.com/Css
www.w3schools.com/Css/css_intro.asp
www.w3schools.com/html/html_css.asp

There are many video tutorials. Here are a few:

What is CSS and How Does it Work? *https://www.youtube.com/watch?v=XPv4EeB0PJ8* Zero to Hero *https://www.youtube.com/watch?v=1Rs2ND1ryYc*
CSS in 2020 a Practical Guide *https://www.youtube.com/watch?v=QG8Zv_doPOM*

Chapter 13

Generics and the API Collection Framework

In this chapter

In this chapter, we extend our knowledge of methods and classes by incorporating the feature of generics into them. This will make the methods and classes we write more reusable and less error prone. A generic method can be passed arguments of different types, and the types of the data members of a generic class can be specified when an instance in the class is created. These powerful features can be used to write classes called data structures that can store, fetch, and process a set of any type of objects. Data structures such as lists, queues, stacks, sets, and hash maps will be discussed.

A set of highly reusable generic methods, interfaces, and classes make up the Java Collection Framework. The methods implement classic computer algorithms, and the classes implement commonly used techniques for efficiently processing large data sets. We will learn the functionality of these methods and classes and how to incorporate them into the programs we write. In addition, we will discuss two groupings of classes included in the framework and the advantages of the Map grouping that can be used to efficiently locate a particular object in a large data set by simply specifying a key value that has been associated with the object.

After successfully completing this chapter, you should:

- Know how generic classes and methods extend reusability and reduce runtime errors
- Understand the difference between a value parameter list and a type parameter list
- Be able to write generic classes and methods and use generic interfaces
- Know how to invoke generic methods and how to declare instances of generic classes
- Understand and be able to write and use overloaded generic methods

- Be able to use the methods in the Java Collection Framework in the programs you write, including the methods defined in the `Collections` class
- Understand the generic interface hierarchy defined in the Java Collection Framework
- Create applications that use the generic data structures classes implemented in the framework
- Understand the differences between Lists, Sets, Queues and Priority Queues, and Maps and their implementations in the framework

13.1 OVERVIEW

A feature of programming languages that extends the reusability of methods and classes is called generics. It provides reusability by permitting the type of a method's parameters and returned value to be specified by the method's invoker and by permitting the type of a class's data members to be specified when an instance of the class is created. Thus, the use of generics makes it possible to write one sort method that can sort an array of *any* type of object passed to the method and to write one class that can store, fetch, delete, and process a set of objects of any type. In addition, the use of generics can also move certain type-checking errors from runtime to compile time, where they are easier to detect and eliminate.

Java supports generics and provides a syntax that can be used to implement methods and classes in a generic way. Using this powerful feature of the language, we can pass any type of argument into a generic method's parameter and specify the type of a generic class's data members and its methods' parameters. In addition, the Java API provides generic implementations of many of the classic algorithms used to efficiently process and store large data sets containing data of any type. These generic implementations are known as *collections*, and they comprise the Java collections framework.

In the first part of this chapter, we will become familiar with Java's implementation of generics, how to write generic methods and classes, and how to invoke generic methods and declare instances of generic classes. This will facilitate our understanding of the second part of the chapter in which we will become familiar with many of the methods, interfaces, and classes in Java's Collection Framework, all of which are implemented using generics.

13.2 GENERIC METHODS

When any method with a non-empty parameter list is invoked, values are passed into each of its parameters. These values are usually different for each invocation of the method, and they can be either primitive values or the address of an object (reference values). The parameter list, enclosed in parentheses at the end of the method's signature, can be thought of as a list of values that will be passed to the method or a *value* parameter list. This list contains the name of each parameter and the type of the value that will be passed to it. When the value passed to the method is a primitive value, the type is a primitive type. When the value passed to the method is a reference value, the address of an object, the type is a class name.

Generic methods are passed values just like non-generic methods. What makes them different from non-generic methods is that at least one of the parameters in their value parameter list can be passed a reference to an instance of *any* class. A specific class name is not coded for this parameter. Instead, a *type placeholder* is coded in the value parameter list as the parameter's class name, and the placeholder is included in a *type* parameter list section of the method's signature.

```
public static <T> void outputAnyObject(T theObject)
```

Type parameter lists are coded just to the left of the method's returned type. They are a list of the type placeholders used in the method's parameter list, separated by commas, and the list is enclosed in angle brackets. For example: `<T>` or `<T1, T2>`. Good coding style dictates that placeholder names begin with a capital letter and be as brief as possible. All of the type placeholders used in a method's value parameter list must be included in the method's type parameter list.

For example, the first signature shown below could be used for a generic method that outputs an instance of any class, and the second signature could be used for a method that outputs two instances of any class:

```
public static <T> void outputAnyObject(T theObject)
public static <T> void outputAnyTwoObjects(T object1, T object2)
```

The type placeholders that appear in a method's signature can be used within the code body of the method. Often, they are used to declare local reference variables within the methods that can reference instances of the class of the parameter of which they are a part.

The signatures of generic methods can also include non-generic types. For example, the following signature could be used for a generic method that outputs an instance of any class a given number of times (i.e., `nTimes`):

```
public static <T> void outputNTimes(T theObject, int nTimes)
```

The following signature could be used for a method that outputs two objects, possibly of two different classes, a given number of times:

```
public static <T1, T2> void outputNTimes2(T1 obj1, T2 obj2, int nTimes)
```

NOTE *The primitive types (`int`, `double`, `char`, etc.) cannot be passed to generic parameter types.*

Only object references can be passed to generic parameter types. To pass a primitive value to a parameter whose type is a generic placeholder, it must be wrapped inside an instance of a primitive wrapper class (e.g., `Integer`, `Double`, `Char`, etc.). As we will see, this can be performed by the autoboxing feature of Java.

Generic Returned Types

In a non-void generic method, one of the type placeholders can be used to designate the returned type. For example:

```
public static <T1, T2> T2 output2Objects(T1 object1, T2 Object1)
```

In this case, T2 is the returned type, and the method would have to return the address of an object whose class is the same as the class of the second argument passed to it. To return an object that is the same type as the first parameter, the returned type would be changed to T1. Generic methods can return primitive values and non-generic reference variable types, just as non-generic methods can, by coding their specific type as the method's return type.

The application GenericParameters shown in Figure 13.1 contains a generic method output2Objects (lines 18–24) that outputs two objects of any type sent to it and returns the object passed to its second parameter. The method is invoked twice within the application. In the first invocation of the method (line 12), the arguments are two primitive variables, amount and price, declared on lines 5 and 6. Because primitives cannot be passed to generic parameters, these two arguments are autoboxed, and then the address of the Integer and Double objects are passed to the method's parameters.

NOTE *Primitive arguments are autoboxed into wrapper objects before they are passed to generic parameters.*

Similar autoboxing is performed on the first argument of the second invocation (line 13), which is a primitive char variable declared on line 7. The second argument in this invocation is an instance of the class Student, shown in Figure 13.2, which is declared on line 8. Before the application ends, lines 14 and 15 of Figure 13.1 output the generic method's returned values. The program's output is shown in Figure 13.3.

The declarations on lines 5–7 of the application could have declared amount, price, and initial to be three wrapper objects and initialized them using the wrapper classes's constructors as shown below. Although the coding of the invocation statements would not change, this would eliminate the need for the autoboxing. The declarations on lines 5–7 are considered to be better programming style.

```
Integer amount = new Integer(45);
Double price = new Double(567.89);
Character initial = new Character('P');
```

The type parameter list in the signature of the generic method output2Objects (line 18) includes two generic placeholders: T1 and T2. These placeholders are used in the method's parameter list as the types of the objects passed to it. The placeholder T2 is also used as the method's returned type because that is the generic type of the method's second parameter that is returned on line 23.

Lines 20 and 21 output the objects passed to the method to the system console using implicit invocations of the toString method. During the first invocation of the generic method (line 12), the toString methods of the Integer and Double classes are invoked because this invocation passes these autoboxed types into the method's type placeholders. Similarly, during the second invocation of the method (line 13), the toString methods of the Character class and Student class (lines 12–16 of Figure 13.2) are invoked. If the Student class did not contain a toString method, the toString method inherited from the class Object would have been invoked.

The object returned on line 12 of Figure 13.1 is an instance of the wrapper class `Double`. Its assignment to the primitive `double` variable `returnedPrice` declared on line 9 is valid because Java's autounboxing feature unwraps the value stored inside the object before the assignment is made.

```
1    public class GenericParameters
2    {
3      public static void main(String[] args)
4      {
5        int amount = 45;
6        double price = 567.89;
7        char initial = 'P';
8        Student s1 = new Student(19, "Sam Jones");
9        double returnedPrice;
10       Student returnedStudent;
11
12       returnedPrice = output2Objects(amount, price);
13       returnedStudent = output2Objects(initial, s1);
14       System.out.println(returnedPrice);
15       System.out.println(returnedStudent);
16     }
17
18     public static <T1, T2> T2 output2Objects(T1 object1, T2 object2)
19     {
20       System.out.println(object1);
21       System.out.println(object2 + "\n");
22
23       return object2;
24     }
25   }
```

Figure 13.1
The application **`GenericParameters`**.

```
public class Student
{
  int age;
  String name;

  public Student(int age, String name)
  {
    this.age = age;
    this.name = name;
  }

  public String toString()
  {
```

```
        String s;
        return s = "age " + age + " name " + name;
    }
}
```

Figure 13.2
The class **Student**.

```
45
567.89

P
age 19 name Sam Jones

567.89
age 19 name Sam Jones
```

Figure 13.3
The output produced by the application **GenericParameters**.

13.2.1 Overloading Generic Methods

As discussed in Chapter 3, overloaded methods are a set of methods defined in a class that have the same name and different parameter lists. When the translator encounters an invocation of an overloaded method in a set of non-generic methods, it seeks a version of that method whose parameters match the invocation's arguments. This version of the method is invoked when the program executes the invocation statement. If a match cannot be found, a translation error is generated.

NOTE *Generic methods can be included in a set of overloaded methods.*

Generic methods can overload non-generic methods and other generic methods. When generic methods are included in a set of overloaded methods, the translator seeks a version of the method that *best fits* the argument list in the method invocation statement. To determine a best fit, the translator follows a set of protocols. The protocols do *not* take into consideration the order in which the overloaded methods are coded. If a best fit cannot be found that is consistent with the protocols, a translation error is generated.

The methods shown in Figure 13.4 can be used to illustrate three of these best-fit protocols. The figure contains four overloaded versions of the method `identifyYourself`, each of which has two parameters. Version 1, shown at the top of the figure, is non-generic. The other three versions are generic. As we will discover, the best-fit protocols do not permit all four of these methods to be included in a set of overloaded methods.

When the method `identifyYourself` is invoked and passed two *integer primitive* arguments, Version1 could be executed because it has two integer parameters. The other three versions of the method could also be executed because one argument (in the case of Version 2) or both of the arguments (in the case of Versions 3a and 3b) could be autoboxed and the `Integer` wrapper object(s) could then be passed to the methods' parameters. The best-fit protocol in this case results in Version 1's execution because the types in its parameter list (`int`) are an exact match with the two integer arguments passed to it.

When the method is invoked and the first argument passed to it is an object and the second is an integer primitive value, Version 2 could be invoked. Version 3a could also be invoked because the second argument could be autoboxed, and the resulting `Integer` wrapper object could then be passed to the method's second parameter. It turns out that Version 3b could also be invoked, even though its parameter list implies that the two objects passed to it must be of the same type, `T`. The best-fit protocol in this case results in Version 2's execution because the type of its second parameter, `int`, is an exact match for the second argument passed to the method.

```
// Version 1
public static void identifyYourself(int a, int b)
{
      System.out.println("Version 1 was invoked");
}

// Version 2
public static <T> void identifyYourself (T a, int b)
{
      System.out.println("Version 2 was invoked");
}

// Version 3a
public static <T1, T2> void identifyYourself (T1 a, T2 b)
{
      System.out.println("Version 3a was invoked");
}

// Version 3b
public static <T> void identifyYourself (T a, T b)
{
      System.out.println("Version 3b was invoked");
}
```

Figure 13.4
A set of overloaded methods.

Best-fit protocols do not allow Versions 3a and 3b to be in the same class because both can accept two object instances of the same type or different types. When coded in the same class, an attempt to invoke `identifyYourself` and pass it two objects results in a translation error, indicating that the invocation is ambiguous (could be serviced by either version of the method). The coding of Version 3a is preferred over Version 3b.

The application `GenericOverloading`, shown in Figure 13.5, includes the first three overloaded versions of `identifyYourself` presented in Figure 13.4. The invocations of the method on lines 5–8 and the output produced by the program (Figure 13.6) demonstrate the translator's best-fit selection protocols. The coding order of the methods has been reversed to demonstrate that the best-fit protocols do not consider the order in which the methods appear in the class.

In the interest of brevity, lines 6–8 pass the method nameless instances of the classes `Integer` and `Double`, and the `Student` class (shown in Figure 13.2).

```
1    public class GenericOverloading
2    {
3      public static void main(String[] args)
4      {
5        identifyYourself(1, 2); //int, int: Version 1
6        identifyYourself(new Integer(10), 2); //object, int: Version 2
7        identifyYourself(2, new Double(20.3)); //int, object: Version 3a
8        identifyYourself(new Integer(10), new Student(19, "Evie")); //V3a
9      }
10
11     // Version 3a
12     public static <T1, T2> void identifyYourself(T1 a, T2 b)
13     {
14       System.out.println("Version 3a was invoked");
15     }
16
17     // Version 2
18     public static <T> void identifyYourself(T a, int b)
19     {
20       System.out.println("Version 2 was invoked");
21     }
22
23     // Version 1
24     public static void identifyYourself(int a, int b)
25     {
26       System.out.println("Version 1 was invoked");
27     }
28   }
```

Figure 13.5
The application **`GenericOverloading`**.

```
Version 1 was invoked
Version 2 was invoked
Version 3a was invoked
Version 3a was invoked
```

Figure 13.6
The output produced by the application **`GenericOverloading`**.

13.2.2 Arrays as Generic Parameters and Returned Values

As is the case for non-generic methods, any parameter in a generic method's parameter list can be a reference to an array object. By specifying the parameter's type to be one of the generic placeholders included in the method's type parameter list followed by an open and closed brace, the address of any array of objects can be passed to the parameter. The following generic method signature could be used for a method that outputs the contents of a non-primitive type array and returns one of its elements.

```
public static <T> T outputArray(T[] anArray)
```

The pair of brackets that follow the placeholder in the method's signature indicate that the address of an array will be passed to the method's parameter.

The application `GenericsArrayParameters`, shown in Figure 13.7, contains a generic method named `outputArray` (lines 24–33) that outputs the contents of the array of objects passed to it to the system console. The method also returns one element of the array whose index is specified by the invoker. Any array to be output can be passed to the method's first parameter, *except* for an array of primitive values. (As previously mentioned, the Java autoboxing feature will not convert an array of primitives passed to the method to an array of wrapper class objects.) The output produced by the program is shown in Figure 13.8.

The method's signature (line 24) contains two parameters, `anArray` and `elementReturned`, and includes one generic placeholder, `T`, in its type parameter list. This placeholder is used as the type of the method's first parameter. It is also used as the method's generic returned type because an element of the array will be returned by the method. The method's `for` loop (lines 26–29) outputs all of the array elements using an implicit invocation of the `toString` method, and line 32 returns the array element whose index is passed to the method's second parameter. The application invokes the method three times, passing it a different array each time.

The first two invocations pass the method the array of wrapped integer values (line 15) and an array of wrapped real numbers (line 16), declared and initialized on lines 5 and 6 respectively. These arrays cannot be declared as arrays of primitive values because Java will not autobox an array of primitive values before passing it to a generic array parameter. The values contained in the returned `Integer` and `Double` objects are autounwrapped and assigned to the primitive variables `intReturned` and `doubleReturned`.

An array of `Student` objects, whose class is defined in Figure 13.2, is passed to the method on line 17. The address of the student object returned from this invocation is assigned to the reference variable `studentReturned` declared on line 10. Lines 19–21 produce the last three outputs, which are the contents of the two returned wrapper objects and the returned `Student` object.

```
1   public class GenericsArrayParameters
2   {
3     public static void main(String[] args)
4     {
5       Integer[] intArray  = {10, 20, 30, 40, 50};
```

```
    Double[] doubleArray = {11.1, 22.2, 33.3, 44.4};
    Student[] studentArray = new Student[2];
    int intReturned;
    double doubleReturned;
    Student studentReturned;

    studentArray[0] = new Student(19, "Sam Jones");
    studentArray[1] = new Student(20, "Nora King");

    intReturned = outputArray(intArray, 3); //autounbox the returned
    doubleReturned = outputArray(doubleArray, 2); //int and double
    studentReturned = outputArray(studentArray, 1);

    System.out.println(intReturned);
    System.out.println(doubleReturned);
    System.out.println(studentReturned);
  }

  public static <T> T outputArray(T[] anArray, int elementReturned)
  {
    for(int i = 0; i < anArray.length; i++)
    {
      System.out.println(anArray[i]);
    }
    System.out.println();

    return anArray[elementReturned];
  }
}
```

Figure 13.7
The application **GenericArrayParameters**.

```
10
20
30
40
50

11.1
22.2
33.3
44.4
```

```
age 19 name Sam Jones
age 20 name Nora King

40
33.3
age 20 name Nora King
```

Figure 13.8
The output produced by the application **`GenericArrayParameters`**.

Returning Generic Arrays

As discussed in Chapter 6, when a non-generic array is returned from a method the type of the array followed by a set of brackets (e.g., `Student[ ]`) replaces the keyword **`void`** in the method's signature, and a return statement that includes the array's name is coded in the method.

The same syntax is used to return a generic array from a generic method, except that the type of the array is replaced with one of the generic placeholders used in the method's signature. For example, the following method signature could be used in a method that returns a generic array whose type was the same as the first argument passed to the method:

```
public static <T1, T2> T1[] returnArray(T1[] anArray, T2 anObject)
```

The array returned would be the one whose name is included in a return statement executed within the method.

The generic method shown in Figure 13.9 swaps the first two elements of the array passed to it and returns the modified array to the invoker. As previously mentioned, the array passed to the method cannot be an array of primitive values. Line 3 of the method creates a generic local variable named `temp` whose type is the type placeholder used to specify the type of the array passed to the method.

```
1     public static <T> T[] swap0and1(T[] anArray)
2     {
3       T temp;
4
5       temp = anArray[0];
6       anArray[0] = anArray[1];
7       anArray[1] = temp;
8
9       return anArray;
10    }
```

Figure 13.9
A generic method that returns an array.

13.2.3 Copying a Generic Array

Although local generic variables can be declared inside a generic method, limits are imposed on the creation of generic arrays inside a generic method. The syntax used to create a non-generic array inside a method, which could be used to hold a copy of another non-generic array, cannot be used to create a generic array. The following declaration produces a compile time *generic array creation* error if `T` is a generic placeholder:

```
T[] copy = new T[100]; //not allowed
```

The good news is that a copy of a generic array can be created within a generic method using the `Arrays` class's `copyOf` method that was discussed in Chapter 6, and the copy can then be modified and returned from the method. Another alternative is that an array-like instance of the class `ArrayList` can be created, modified, and returned from a generic method. The `ArrayList` class will be discussed later in this chapter (Section 13.3.1), as will the correct syntax for creating a generic array that is not a copy of another generic array.

Lines 29–39 of the application `ReturningGenericArrays`, shown in Figure 13.10, is a generic method named `invertArray` that copies any type of array passed to it and then returns a modified version of the array to the method's invoker. The returned array contains the elements of the array passed to the method with their order reversed (first to last becomes last to first). The output produced by the program is shown in Figure 13.11.

Line 31 of the method declares a generic array reference variable named `copy` using the placeholder `T` that appears in the method's signature (line 29). The `Arrays` class's `copyOf` method is used on line 33 to create a duplicate of the array passed to the method and assign its address to the variable `copy`. The code of the `for` loop that begins on line 34 then copies the object references from the original array (`anArray`) into the newly created array (`copy`) in reverse order. Line 38 returns the array created inside the method to the invoker.

The first time the method is invoked (line 14), it is passed the `Integer` wrapper array declared on line 7. The second invocation (line 15) passes the method the array of objects declared on lines 9–12. This array is created by initializing it to three nameless `Student` objects whose class is shown in Figure 13.2.

The arrays returned from the method invocations are output inside the two `for` loops that begin on lines 17 and 23. The first of these loops outputs the contents of the original and reverse order `Integer` arrays, `iArray` and `iArrayReturned`, side by side (top part of Figure 13.11). The second loop repeats this process for the `Student` arrays `sArray` and `sArrayReturned` (bottom part of Figure 13.11).

```
1   import java.util.Arrays;
2
3   public class ReturningGenericArrays
4   {
5     public static void main(String[] args)
6     {
7       Integer[] iArray= {1,2,3,4};
```

```
      Integer[] iArrayReturned;
      Student[] sArray = {new Student(17, "Robert"),
                          new Student(20, "Carol"),
                          new Student(16, "Maggie")};
      Student[] sArrayReturned;

      iArrayReturned = invertArray(iArray);
      sArrayReturned = invertArray(sArray);

     for(int i = 0; i < iArray.length; i++) //all the Integer Objects
      {
        System.out.println(iArray[i] + "\t" + iArrayReturned[i]);
      }
      System.out.println();

     for(int i = 0; i < sArray.length; i++) //all the Student Objects
      {
        System.out.println(sArray[i] + "\t" + sArrayReturned[i]);
      }
    }

    public static <T1> T1[] invertArray(T1[] anArray)
    {
      T1[] copy;

      copy = Arrays.copyOf(anArray, anArray.length);
      for(int i = 0; i < copy.length; i++)
      {
        copy[i] = anArray[copy.length - 1 - i];
      }
      return copy;
    }
  }
```

Figure 13.10
The application **ReturningGenericArray**.

```
1   4
2   3
3   2
4   1

age 17 name Robert  age 16 name Maggie
age 20 name Carol   age 20 name Carol
age 16 name Maggie  age 17 name Robert
```

Figure 13.11
The output produced by the application **ReturningGenericArrays**.

13.2.4 Operating on Generic Objects

As we have already seen, one way to perform processing on an object is to invoke a worker method. For example, to fetch the private integer data member named `x` of an instance of a `Snowman` object named `s1`, we could invoke the class's `getX` method on the object to perform the work of fetching the variable's contents:

```
Snowman s1 = new Snowman();
int x = s1.getX();
```

When the translator processes this invocation, it searches the object's class and its inheritance chain for a method named `getX` that has an empty parameter list and returns an integer. If it finds a method with this signature, the translation continues. Otherwise, the translation ends in a *cannot find symbol method getX( )* error.

Now consider the case when the `s1` is a generic parameter in the signature of a generic method, and the invocation of the `getX` method is issued from within the generic method, as shown in this code fragment:

```
public static <T> boolean collision(T s1, T s2)
{
  int x1 = s1.getX()
          :
}
```

In this case, the translation of the method will end in a *cannot find symbol method getX( )* translation error, even if the class of the argument passed to `s1` contains a `getX` method. The object `s1`'s class is specified to be the generic type placeholder `T` in the method's parameter list, so now there is no relationship between the parameter `s1` and the `Snowman` class, or any other class. The translator cannot look into `T` to locate the method `getX`; it is simply a generic placeholder.

To remedy this problem, the author of the generic method `collision` would include an `extends` clause inside the type parameter list of the generic method's signature that included the name of an interface that defines the `getX` method's signature. Assuming the name of the interface is `Detectable`, the modified signature of the generic method would be:

```
public static < T extends Detectable <T> > boolean collision(T s1, T s1)
```

The `extends` clause added to the method's signature directs the translator to look into the interface `Detectable` to verify the `getX` method's signature, and only objects whose classes implement this interface can be passed to this method.

NOTE *The keyword* **`extends`** *is always used in a type parameter list to identify an interface. The keyword* **`implements`** *is not used.*

The class of any object passed to the method would have to implement the interface `Detectable(` i.e., include an `implements` clause in its heading, and an implementation of a `getX` method whose signature is defined in that interface). If it did not implement the interface, the translator would issue the error message. In the case when two `Snowman` objects were passed to the method, the error message would indicate that the method *could not be applied to (Snowman, Snowman)*.

The application `OperatingOnGenericObjects` shown in Figure 13.12 contains a generic method named `min` (lines 15–26) that returns the address of the smallest object in an array of objects passed to it. The work of comparing two elements of the array is performed by a worker method named `compareTo`, which is invoked on an element of the generic array (line 20) and passed a reference (defined on line 17) to an element of the array. An `extends` clause involving the array's generic placeholder `T` has been added to the method's signature (line 15) to permit the translator to look into an interface named `Comparable` to verify the signature of the `compareTo` method invoked on line 20. The syntax `<T>` that follows the name of the interface specifies that an object in the class `T` will be passed to the `compareTo` method. The result is that only objects whose classes implement this interface can be passed to the method `min`, and the object passed to the `compareTo` method on line 12 must be a reference to the type of object passed to the method `min`.

Several API classes including the `String` class and the primitive wrapper classes, which include the `Integer` class, implement the interface `Comparable`. It is left up to the implementer of the interface to decide what it means for an object in the implementing class to be equal to, greater than, or less than another instance of the class. As would be expected, the `Integer` class compares two `Integer` objects numerically.

The class `StudentV2` (shown in Figure 13.13) also implements the interface `Comparable`. The inclusion of `<StudentV2>` at the end of the `implements` clause in the class's heading designates that its version of the `compareTo` method must be passed a `StudentV2` instance. Line 20 effectively compares the ages of two instances of the class numerically.

The method `min`, invoked on line 11 of Figure 13.12, is passed the array of `Integer` objects declared on line 5, and the returned minimum object is then output to the system console (Figure 13.14). During this invocation of the method, the `Integer` class's implementation of `compareTo` is invoked on line 20 because the type of `anArray` is `Integer`. In the second invocation of the method `min` (line 12), the method is passed the `StudentV2` array, defined on lines 6–9 of Figure 13.12, which causes line 20 to invoke the `compareTo` method on lines 18–21 of Figure 13.13. The returned minimum `StudentV2` object returned from `min` is then output to the system console (Figure 13.14).

```
1   public class OperatingOnGenericObjects
2   {
3     public static void main(String[] args)
4     {
5       Integer[] iArray= {110, 36, 78, 43, 23, 83, 34, 24};
6       StudentV2[] sArray = {new StudentV2(18, "Sam"),
7                             new StudentV2(32, "Carol"),
8                             new StudentV2(16, "Maggie"),
9                             new StudentV2(25, "James")};
10
11      System.out.println("iArray minimum is: " + min(iArray));
12      System.out.println("sArray minimum is: " + min(sArray));
13    }
14
```

```
  public static <T extends Comparable<T>> T  min(T[] anArray)
  {
    T minimum = anArray[0];
    for(int i = 1; i < anArray.length; i++)
    {
      if(anArray[i].compareTo(minimum) < 0)
      {
        minimum = anArray[i];
      }
    }
    return minimum;
  }
}
```

Figure 13.12
The application **OperatingOnGenericObjects**.

```
public class StudentV2 implements Comparable<StudentV2>
{
  private int age;
  private String name;

  public StudentV2(int age, String name)
  {
    this.age = age;
    this.name = name;
  }

  public String toString()
  {
    String s;
    return s = "age " + age + " name " + name;
  }

  public int compareTo(StudentV2 s1)
  {
    return age - s1.age;
  }
}
```

Figure 13.13
The class **StudentV2**.

```
iArray minimum is: 23
sArray minimum is: age 16 name Maggie
```

Figure 14.14
The output from the application **OperatingOnGenericObjects**.

13.3 GENERIC CLASSES

A generic class is a class whose heading contains a type parameter list, which is used to specify the type of one or more of its data members. They are widely used in the implementation of data structures. The class can contain both non-generic and generic methods that use the type placeholders included in the class's type parameter list. The type parameter list is coded immediately after the class's name. For example:

```
public class AGenericClass <T1, T2, T3>
```

If a generic class extends another class and/or implements an interface, the `extends` and `implements` clauses are added to the method's heading after its type parameter list. For example, the heading of a generic class named `Employee` that was a subclass of `Person` could be:

```
public class Employee <T> extends Person implements Comparable<Employee>
```

The following statement declares an instance of this class using its two parameter constructor:

```
Employee s1 = new Employee <Integer> (45323, "Ryan");
```

The `<Integer>` included in the declaration is called a *type argument list*. A type argument list is enclosed in angle brackets and consists of a list of one or more class names separated by commas (e.g., `<Integer`, `String`, `Integer>`). It is coded immediately before the arguments passed to the class's constructor. The type argument list must include one class name for each type parameter included in the class's heading.

When an instance of a generic class is declared, the class names in the type argument list of the declaration are matched with the type parameters in the class's heading, one for one in the order in which they appear. These class names are effectively substituted for the type placeholders wherever they are used in the class's code. For example, when the object `s1` is declared on line 3 of the `main` method shown at the top of Figure 13.15, the class `String` is effectively substituted for the placeholder `T` on lines 3 and 6 of the class `AStudentV3` shown in the bottom portion of the figure. As a result, `s1`'s data member `id` is a reference to a `String` object, and the first parameter passed to the class's constructor on line 3 of the `main` method must be a `String` (*CS103* on line 3 of `main`).

Similar substitutions are made when the object `s2` is declared on line 4 of the `main` method. Because the type argument list on that line contains the class `Integer`, `s2`'s data member `age` will be a reference to an `Integer` object, and the first argument passed to the class's constructor on line 4 of the `main` method must be an instance of an `Integer`. In this case, it is the nameless `Integer` wrapper containing a `10`. The integer literal `10` could be substituted for the nameless object passed to the constructor on line 4, because it would be autoboxed before it was passed to the constructor's first parameter.

```
1   public static void main(String[] args)
2   {
3     AStudentV3 s1 = new AStudentV3 <String>("CS103", "Tom");
4     AStudentV3 s2 = new AStudentV3 <Integer>(new Integer(10), "Ryan");
5   }
```

```
public class AStudentV3 <T>
{
  private T id;
  private String name;

  public StudentV3(T id, String name)
  {
    this.id = id;
    this.name = name;
  }
}
```

Figure 13.15
A generic class and a **`main`** method that declares two instances of the class.

When an instance of a generic class is declared, a type argument list should always be included in the declaration immediately before the arguments passed to the class's constructor, as shown on lines 3 and 4 of the `main` method in Figure 13.15. Its inclusion provides Java type checking and is considered good programming practice.

The type argument list can also be included between the class name and the variable name on the left side of an object declaration statement. When used, this list must match the argument list that appears on the right side of the declaration. For example, lines 3 and 4 of the `main` method shown in Figure 13.15 would become:

```
AStudentV3 <String> s1 = new AStudentV3 <String>("CS103", "Tom");
AStudentV3 <Integer> s2 = new AStudentV3 <Integer>(new Integer(10),"Ryan");
```

The inclusion of the argument list in front of the variable names effectively extends the type checking Java performs. A subsequent attempt to assign the reference variable `s1`, whose `id` data member was specified to be a string, to `s2`, whose `id` data member is an instance of an `Integer` (e.g., `s2 = s1`), results in an *incompatible types* translation error. This level of translation-time type checking is usually desirable. Although the following two declarations are also syntactically correct, they are considered unsafe from a type-checking viewpoint.

```
//******** Unsafe generic object declarations *********//
AStudentV3 <String> s1 = new AStudentV3("CS103", "Tom");
AStudentV3 s1 = new AStudentV3("CS103", "Tom");
```

The application `GenericClasses`, shown in Figure 13.16, declares four type-safe instances of the generic class `StudentV3` shown in Figure 13.17. The output produced by the program is shown in Figure 13.18. Each `StudentV3` object has an identification (ID) number and a name. The type of the ID is declared to be generic on line 3 of Figure 13.17 by coding the placeholder `T`, included in the class's heading, as the type of the data member `id`. The class's constructor also uses the type parameter `T` as the type placeholder for its first parameter.

The three object declarations on lines 6–8 of Figure 13.16 use the preferred type-safe two-argument list syntax to declare one object with a `String` type ID (line 6), and two `Integer` type ID objects (lines 7 and 8). A fourth object is declared on line 9 using the type-safe one-argument list syntax. The types of the first argument passed to the constructors invoked on lines 6–9 are consistent with the class names these lines pass to the `StudentV3` class's type parameter list (when autoboxing is considered). The four objects are output on lines 13–16 using an implicit invocation of the generic class's `toString` method (lines 11–15 of Figure 13.17). When lines 14–16 of Figure 13.16 execute, the `Integer` class's `toString` method is invoked implicitly on line 14 of Figure 13.17 to add the IDs of objects `s2`, `s3`, and `s4` to the string `s`.

The reference variable `s5` declared in line 10 of Figure 13.16 can only reference a `StudentV3` object whose `id` data member is a string because its declaration includes a `String` type parameter. The reference variable `s5` declared on line 11 can reference any `StudentV3` object because it does not include a type parameter list. Type-safe assignments are performed on these variables on lines 18 and 19, and then the objects they reference are output on lines 20 and 21.

Line 23 of Figure 3.16 invokes the `StudentV3` class's `compareTo` method to compare the object `s5` references to the object `s6` references. This method (lines 16–19 of Figure 13.16) compares the `name` data members of two `StudentV3` instances: the object that invoked it and the object passed to its parameter. To make the comparison, the method invokes a `compareTo` method on line 18. Because the name data members used in the invocation are strings, the `String` class's `compareTo` method is invoked. The returned value is seven because the first letter in `s5`'s name (the *T* in *Tom*) is seven characters beyond the first letter of `s6`'s name (the *M* in *Maggie*). This retuned value is the last output shown in Figure 13.17.

```
1    public class GenericClasses
2    {
3      public static void main(String[] args)
4      {
5        Integer id = new Integer(1672);
6        StudentV3 <String> s1 = new StudentV3 <String>("Sci103", "Tom");
7        StudentV3 <Integer> s2 = new StudentV3 <Integer>(1672,"Maggie");
8        StudentV3 <Integer> s3 = new StudentV3 <Integer>(45323, "Ryan");
9        StudentV3 s4 = new StudentV3 <Integer>(53812, "Logan");
10       StudentV3 <String> s5 = null;
11       StudentV3 s6 = null;
12
13       System.out.println(s1);
14       System.out.println(s2);
15       System.out.println(s3);
16       System.out.println(s4 + "\n");
17
18       s5 = s1; //Safe
```

```
    s6 = s2; //Safe
    System.out.println(s5);
    System.out.println(s6);

    System.out.println(s5.compareTo(s6));
  }
}
```

Figure 13.16
The application **GenericClasses**.

```
public class StudentV3 <T> implements Comparable<StudentV3>
{
  private T id;
  private String name;

  public StudentV3(T id, String name)
  {
    this.id = id;
    this.name = name;
  }
  public String toString()
  {
    String s;
    return s = "ID: " + id + "; Name: " + name;
  }
  public int compareTo(StudentV3 s)
  {
    return name.compareTo(s.name);
  }
}
```

Figure 13.17
The class **StudentV3**.

```
ID: Sci103; Name: Tom
ID: 1672; Name: Maggie
ID: 45323; Name: Ryan
ID: 53812; Name: Logan

ID: Sci103; Name: Tom
ID: 1672; Name: Maggie
7
```

Figure 13.18
The output produced by the application **GenericClasses**.

13.3.1 Generic Data Structure Classes

A data structure is an object that can store a larger set of objects, such as 10,000 `Student` objects, in a way that facilitates the operations that will be performed on them. Common operations are fetching and updating an object. An array is a data structure that is part of every programming language, and we have used this data structure to store objects such as `Snowmen` and strings. Other common data structures are stacks, queues, linked lists, trees, and hashed structures. These data structures are usually implemented as generic classes so that, like an array, they can be used to store a set of any type of object. In this section, we use the data structure queue to illustrate the nuances of implementing a generic data structure class.

A queue can be thought of as a fair waiting line that has a *front or head* and *rear or tail* end. When an object is added to a queue, it is added at the end, or rear, of the queue. When an object is fetched from the queue, the object at the front of the queue is fetched *and* deleted from the queue. This process of adding and fetching objects is referred to as a First In First Out (FIFO) process: the first object added to the data structure is the first object fetched (and deleted) from the data structure. The add operation is called `enqueue` (*en*ter the *queue*), and the fetch/delete operation is called `dequeue` (*de*part from the *queue*).

Queues are used by applications that process objects, once and only once, in the order they are received. Print requests to a shared printer are stored in a queue, as is information about airplane objects waiting for their turn to land on a busy runway, as are processes waiting to be run by an operating system.

A Non-generic Queue

The class `Queue` shown in Figure 13.19 is a non-generic array-based implementation of a queue that can only store `StudentV4` objects whose class is shown in Figure 13.20. The class `Queue` contains a constructor (lines 11–15), an `enQueue` method (lines 17–30), and a `deQueue` method (lines 32–46). The implementation of these methods make the class a *circular* queue because lines 27 and 42 reposition the front and rear of the queue back to zero to prevent them from exceeding the bounds of the array. The application `NonGenericQueueApp`, shown in Figure 13.21, declares a `Queue` object (line 5) and then adds (enqueue), fetches/deletes (dequeue), and outputs several `StudentV4` objects. The output produced by the program is shown in Figure 13.22.

The queue implementation shown in Figure 13.19 is array based. It stores the object passed to the `enqueue` method on line 17 in an array of `StudentV4` objects named `data`, declared on lines 9 and 14. The `dequeue` method returns an element of this array on line 44.

The size of the array is passed to the class's constructor (line 11) and stored in the data member `size` (line 13). The `enqueue` method returns `false` (line 21) when the queue is full, as determined by line 19, and the `dequeue` method returns `null` (line 37) when the queue is empty, as determined by line 35. The remaining implementation details are not relevant to our discussion and are typically discussed in a data structures textbook.

The application shown in Figure 13.21 creates an instance of a `Queue` on line 5 that can store a maximum of four `StudentV4` objects. Then it creates four objects and invokes the `enqueue` method to store them in the queue (lines 8–15). An attempt to store a fifth object in the queue (lines 16–18) returns `false` (the first output shown in Figure 13.22, produced by lines 17–18). The other five outputs shown in Figure 13.22 are produced by the five invocations of the `dequeue` method inside the `for` loop that begins on line 20 of Figure 13.21. The fifth invocation returns `null` because the queue is empty.

```java
// A Non-Generic Queue. It can only queue StudentV4 Objects

 public class Queue
 {
   private int size;
   private int numOfNodes = 0;
   private int front = 0;
   private int rear = 0;
   private StudentV4[] data;

   public Queue(int n)
   {
     size = n;
     data = new StudentV4[n];
   }

  public boolean enQueue(StudentV4 newItem) //add a StudentV4 object
   {
     if(numOfNodes == size) //the queue is full
     {
       return false;
     }
     else //add the object to the structure
     {
       numOfNodes = numOfNodes + 1;
       data[rear] = newItem;
       rear = (rear + 1) % size;
       return true;
     }
   }

   public StudentV4 deQueue( ) //fetch and delete a StudentV4 object
   {
     int frontLocation;
     if(numOfNodes == 0) //the queue is empty
     {
       return null;
     }
     else //return an object from the structure
     {
       frontLocation = front;
       front = (front + 1) % size;
```

```
      numOfNodes = numOfNodes - 1;
      return data[frontLocation];
    }
  }
}
```

Figure 13.19
The class **Queue**.

```
public class StudentV4 implements Comparable <StudentV4>
{
  private int id;
  private String name;

  public StudentV4(int id, String name)
  {
    this.id = id;
    this.name = name;
  }

  public String toString()
  {
    String s;
    return s = "ID: " + id + "; Name: " + name;
  }

  public int compareTo(StudentV4 s)
  {
    return name.compareTo(s.name);
  }
}
```

Figure 13.20
The class **StudentV4**.

```
public class NonGenericQueueApp
{
  public static void main(String[] args)
  {
    Queue aQueue = new Queue(4);
    StudentV4 aStudent;

    aStudent = new StudentV4(1, "Nora");
    aQueue.enQueue(aStudent);
    aStudent = new StudentV4(2, "Logan");
    aQueue.enQueue(aStudent);
    aStudent = new StudentV4(3, "Evie");
    aQueue.enQueue(aStudent);
    aStudent = new StudentV4(4, "Ryan");
```

```
        aQueue.enQueue(aStudent);
        aStudent = new StudentV4(5, "Skyler"); //queue already full
        System.out.println("Fifth enqueue successful? " +
                           aQueue.enQueue(aStudent));

        for(int i=1; i <= 5; i++) //one more than the queue's capacity
        {
          System.out.println(aQueue.deQueue());
        }
      }
    }
```

Figure 13.21
The application **NonGenericQueueApp**.

```
Fifth enqueue successful? false
ID: 1; Name: Nora
ID: 2; Name: Logan
ID: 3; Name: Evie
ID: 4; Name: Ryan
null
```

Figure 13.22
The output produced by the application **NonGenericQueueApp**.

A Generic Queue Implementation

When implementing a generic data structure, one that can store objects of any class, it's a good idea to implement it as a non-generic version of the structure, such as the implementation shown in Figure 13.19, then, after it is tested and verified, convert it to a generic implementation. This approach is consistent with the concept of divide and conquer.

The class `GenericQueue`, shown in Figure 13.23, is the generic version of the class `Queue` shown in Figure 13.19. The code in the two figures can be compared line by line to find the changes made to produce the generic version, indicated by the yellow highlights in Figure 13.23. The first step in this conversion process is to add a generic parameter list `<T>` to the generic version's heading (line 3 of Figure 13.19) and then use this as a placeholder to eliminate the occurrences of the class name `StudentV4` from the class. This class name appears on lines 9, 14, 17, and 32 of the non-generic version. The new versions of lines 17 and 32 in Figure 13.23 simply substitute the generic placeholder `T` for the class name.

Lines 9 and 14 allocate the array `data` in the non-generic version. There are two options here, neither of which is as obvious as the changes made on to lines 17 and 32. The complication stems from the fact that Java does not permit the declaration of a generic array.

The most obvious change to the two lines, which is shown below, is not valid because Java does not support the use of a generic placeholder in the creation of an array. The new version of line 14 produces a *generic array creation* translation error.

```
9    private T[] data;
14   data = new T[n] //generic array creation is not allowed
```

Because all Java classes inherit from the class `Object` and since polymorphism allows parents to point to children, the following innovative change to line 14 eliminates the *generic array creation tra*nslation error:

```
9    private T[] data;
14   data = (T[]) new Object[n]  //object can point to any class instance
```

The coercion on line 14 is necessary because line 9 declares data as a reference to an array of type `T`. We could proceed in this way and complete the conversion with the following change to line 26, but the changes made to line 14 are considered type-unsafe:

```
26 data[rear] = (T) newItem;
```

The better type-safe approach is to substitute an instance in the API generic class `ArrayList` for the array declared on lines 9 and 14. Taking this approach, the new versions of these lines are shown on lines 9 and 14 of Figure 13.23. Because the variable `data` now references an `ArrayList` object, the invocations of this class's `add` and `get` methods replace the array element accesses on lines 26 and 44 of the non-generic version of the queue in the type-safe conversion of the class shown in Figure 13.23. These two changes complete the generic conversion of the class `Queue`.

```
1    import java.util.ArrayList;
2
3    public class GenericQueue <T>
4    {
5      private int size;
6      private int numOfNodes = 0;
7      private int front = 0;
8      private int rear = 0;
9      private ArrayList <T> data;
10
11     public GenericQueue(int n)
12     {
13       size = n;
14       data = new ArrayList <T> (size);
15     }
16
17     public boolean enQueue(T newItem)
18     {
19       if(numOfNodes == size) //the queue is full
20       {
21         return false;
22       }
23       else //add the object to the structure
24       {
25         numOfNodes = numOfNodes + 1;
```

```
        data.add(rear, newItem);
        rear = (rear + 1) % size;
        return true;
      }
    }

    public T deQueue( ) //fetch and delete an object
    {
      int frontLocation;
      if(numOfNodes == 0) //the queue is empty
      {
        return null;
      }
      else
      {
        frontLocation = front;
        front = (front + 1) % size;
        numOfNodes = numOfNodes - 1;
        return data.get(frontLocation);
      }
    }
  }
```

Figure 13.23
The class **`GenericQueue`**.

The application `GenericQueueApp` shown in Figure 13.24 is the same application presented in Figure 13.21, except that it declares a generic queue object on line 5. Both of these applications produce the same output, which is shown in Figure 13.22. The use of the type argument list at the beginning of the declaration on line 5 of Figure 13.24 ensures that only `StudentV4` instances will be stored in the queue. An attempt to enqueue another type object into the queue, `aQueue`, declared on that line, will result in a *cannot find symbol method enqueue* translation error.

Because the class `GenericQueue` is generic, the application could have declared a second instance of this class to queue 100 `Snowman` objects using the following declaration:

```
GenericQueue <Snowman> snowmanQueue = new GenericQueue <Snowman> (100);
```

```
1   public class GenericQueueApp
2   {
3     public static void main(String[] args)
4     {
5       GenericQueue <StudentV4> aQueue = new GenericQueue <StudentV4> (4);
6       StudentV4 aStudent;
7
8       aStudent = new StudentV4(1, "Nora");
9       aQueue.enQueue(aStudent);
10      aStudent = new StudentV4(2, "Logan");
```

```
        aQueue.enQueue(aStudent);
        aStudent = new StudentV4(3, "Evie");
        aQueue.enQueue(aStudent);
        aStudent = new StudentV4(4, "Ryan");
        aQueue.enQueue(aStudent);
        aStudent = new StudentV4(5, "Skyler"); //queue already full
        System.out.println("Fifth enqueue successful? " +
                           aQueue.enQueue(aStudent));

        for(int i=1; i <= 5; i++) //one more that the queue's capacity
        {
          System.out.println(aQueue.deQueue());
        }
      }
    }
```

Figure 13.24
The application **GenericQueueApp**.

13.4 THE API COLLECTIONS FRAMEWORK

Programs often process information that is comprised of a collection of instances of one class. For example, a process that stores the maintenance work orders for an apartment complex, stores transcripts of all of the students at a college, or searches employee personnel records for a particular employee's phone number. To facilitate the storage and processing of information groups such as these (i.e., work orders, transcripts, and personnel records), the Java API provides *collection classes*. A single instance of one of these classes, a *collection object*, can store an entire information group.

The class `GenericQueue`, shown in Figure 13.23, is an example of a generic collection class. One instance of this class could store all of the maintenance work orders for an apartment complex, a second instance could store the transcripts of all of the students at a college, and a third instance could be used to store a company's employee records. Like our `GenericQueue` class, the API collection classes are generic, and they are therefore highly reusable.

While a queue collection object would be a perfect choice for storing work requests because of its first-in-first-out characteristic, this same characteristic (especially the deletion associated with first-out) makes a queue instance a poor choice for a transcript collection or an employee record collection. In recognition of the fact that any one collection class is not ideally suited for all applications, the Java API implements a variety of the most useful types of collection classes. This variety includes a set, priority queue, linked list, hash map, and several other collection-class implementations. These classes are part of the API *Collections Framework*. The framework also includes:

- A set of interfaces that define the generic signatures of methods common to groups of collection classes, such as the methods `add` and `remove` that add an item to and delete an item from a collection
- A set of static generic methods contained in the class `Collections` that implement algorithms that efficiently perform common operations on collections such as `sort`, `binarySearch`, `min`, and `max` whose names imply their functionality

13.4.1 Framework Interfaces

There are eight core collection interfaces, shown in blue in Figure 13.25, that are divided into two groups: those that extend the interface `Map` and those that extend the interface `Collection`. The core interface `Map` is not a sub-interface; it does not extend another interface. The other seven core interfaces are sub-interfaces; they directly extend an interface.

The significance of the inheritance chains shown in the figure is that a class that implements one of the core interfaces must implement all of the methods whose signatures are contained in it and in its parent interfaces. In addition, any method that can operate on an instance of a class that implements a parent interface can also operate on an instance of a class that implements one of the parent's child interfaces.

All of the core interfaces are generic. As shown next to their names in Figure 13.25, they include one or two type parameters in their type parameter lists to represent the type of the information stored in the collection. The type parameter list of the interface `Collection`, and all of its sub-interfaces, contains one type parameter. An object stored in the API collection classes that implement these interfaces is called an *element*. The type parameter lists of the interfaces `Map` and `SortedMap` include two parameter types. The object pairs stored in a collection class that implements `Map` or `SortedMap` are called a *key* and a *value*.

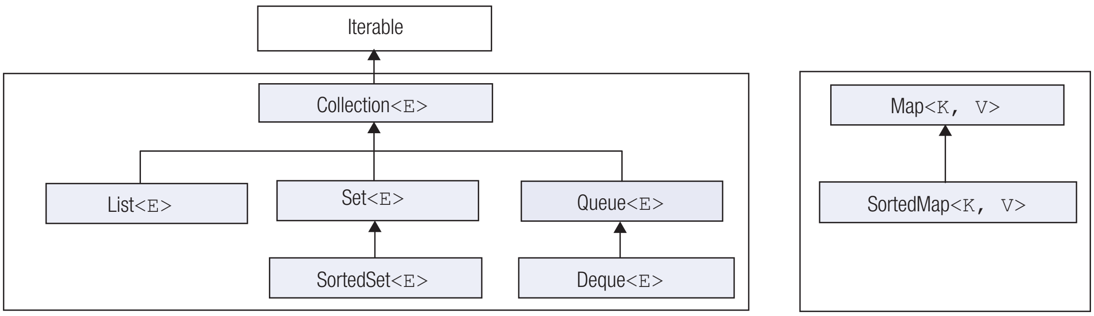


Figure 13.25
The core interfaces of the Java collections framework (shown in blue).

NOTE *Classes that implement the core framework interfaces are called collection classes*

13.4.2 Framework Algorithms: The `Collections` Class

The `Collections` class implements *algorithms* that efficiently perform common operations on the objects contained in a collection. The class contains 52 static methods, most of which are passed an instance of a class that implements a specific framework core interface or an extension of one of these interfaces. Some of the frequently used methods are shown in Table 13.1. The methods in the top portion of the table can only be invoked on objects whose class implements the interface `List`. The methods in the bottom portion of the table can be invoked on objects whose class implements the interface `Collection`. The use of these methods will be demonstrated within the applications and classes presented in the remainder of this chapter.

Table 13.1
Collections Class Methods

Method	Description
Operate on collection objects whose class implements `List`	
`binarySearch`	Returns the index of a specified element
`copy`	Copies one list's elements into another
`indexOfSubList`	Returns the index of the first occurrence of a specified sublist of elements
`replaceAll`	Replaces all occurrences of a specified element with a specified element
`sort`	Sorts the list's elements using their overridden `compareTo` method
`swap`	Swaps the position of two elements whose indices are specified
Operate on a collection object whose class implements `Collection`	
`addAll`	Adds all specified elements, or all elements of an array, to a collection
`disjoint`	Determines if two collections contain no elements in common
`frequency`	Determines the number of occurrences of a specified element
`max`	Returns the largest of the elements using their overridden `compareTo` method
`min`	Returns the smallest of the elements using their overridden `compareTo` method

13.4.3 The `LinkedList` and `ArrayList` Classes

The API Framework classes `LinkedList` and `ArrayList` implement the `List` interface. An instance of a class that implements this interface can contain duplicate objects in its collection. In addition, the objects in the collection have a *sequential ordering* imposed upon them from zero to one less than the number of elements in the collection. As a result, objects stored in a class that implements the interface `List` can easily be processed sequentially based on their location in the list. The `ArrayList` and `LinkedList` classes implement many of the same interfaces, so they share many of the same methods, including the ability to be operated upon by the methods in the `Collections` class.

Objects are added to a collection class that implements the interface `List` by passing them to the `add` method defined in the interface. The one-parameter version of the method appends the object to the collection. The two-parameter version of the method is passed an object and an integer (index), which becomes the new element's location in the collection. The location of the element previously at that position, and the locations of all of the elements beyond it, are increased by one. The `add` method effectively inserts the new element in between two existing elements. Instances of the `ArrayList` class and the `LinkedList` class expand to accommodate the number of elements added to the collection, and elements are not deleted when they are fetched.

Having already used the `ArrayList` class in the implementation of our generic queue at the end of Section 13.3.1 (Figure 13.23), we will use an instance of `LinkedList` in the remainder of this section to become more familiar with many of the methods these two classes share. The following code fragment adds three `StudentV4` objects to the `LinkedList` instance `underGrads`:

```
// Create a linked list and add the elements to it
LinkedList <StudentV4> underGrads = new LinkedList<StudentV4>();
StudentV4 s1 = new StudentV4(2071, "Dana");
StudentV4 s2 = new StudentV4(8129, "Annie");
StudentV4 s3 = new StudentV4(6142, "Nadia");

undergrads.add(s1);
undergrads.add(s2);
undergrads.add(1, s3); //s3 is added in between s1 and s2
```

The `get` and `remove` methods are passed an integer, which is the location of the element to be fetched or removed from the collection. When an element is removed, the locations of all of the elements beyond it are decreased by one. An overloaded version of the `remove` method is passed a reference to the object to be removed, and the `getLast` method returns a reference to the element with the highest index. The last two lines of the following code fragment remove `s3` from the linked list `underGrads` and outputs object `s2`'s information twice.

```
// Delete and fetch elements of a linked list
LinkedList <StudentV4> underGrads = new LinkedList<StudentV4>();
StudentV4 s1 = new StudentV4(2071, "Dana");
StudentV4 s2 = new StudentV4(8129, "Annie");
StudentV4 s3 = new StudentV4(6142, "Nadia");

undergrads.add(s1);
undergrads.add(s2);
undergrads.add(1, s3); //s3 is added in between s1 and s2

undergrads.remove(1); //deletes s3 from the linked list
System.out.println(undergrads.get(1)); //fetches the element 1, now s2
System.out.println(undergrads.getLast()); //fetches the last element, s2
```

The application `LinkedListApp`, shown in Figure 13.26, demonstrates the use of a `LinkedList` generic collection object to store and output student transcript objects. The transcripts are instances of the class `Transcripts` shown in Figure 13.27. Each transcript contains three data members: `name`, `gpa`, and `creditsEarned`, defined on lines 3–5 of that figure. The application also demonstrates the use of many of the `Collections` class's methods listed in Table 13.1 and the use of an iterator, which is a time-efficient way of sequentially processing all of the elements in a `LinkedList`. The output produced by the program is shown in Figure 13.28.

Line 10 of the application shown in Figure 13.26 is a type-safe declaration of a genetic `LinkedList` collection object named `underGrads` that can store a collection of `Transcripts` objects. Any attempt to add anything other than a `Transcript` instance to this collection results in a translation error. The objects created on lines 11–14 are added to the collection using the `LinkList` class's `add` method on lines 16–19. Because the overloaded version of the method that is passed a specific location at which to insert the elements is not used, the new elements, `t1`, `t2`, `t3`, and `t4`, occupy locations zero through three respectively.

Line 25 outputs the objects by invoking the `LinkList` class's `get` method inside the `for` loop that begins on line 23. The method is passed the loop variable, `i`, as the location of the element to be fetched. The annotation produced by line 22, and the four lines of output produced by line 25, are shown at the top of Figure 13.28. The `LinkLink` class's `size` method is used in the `for` statement on line 23 to terminate the loop.

Lines 28–57 use the `Collections` class's static methods to process the elements of the list. The elements of the collection of `Transcript` are sorted by invoking the `Collections` class's `sort` method on line 30 and passing it the collection object `underGrads`. The `sort` method orders and relocates the elements within the collection object's first four locations, then lines 31–34 output them in element-location order (the second group of outputs shown in Figure 13.28). Examining these lines, we see that the transcripts have been sorted in ascending order based on the value of the students' GPA.

The API documentation of the `Collections` class indicates that its `sort` methods sorts in "ascending order, according to the *natural ordering* of its elements." This is another way of stating that the `sort` method invokes the `compareTo` method defined in the interface `Comparable` as part of its sorting algorithm, and it is left to the designer of the class of the list's elements (in our case, the class `Transcripts)` to decide what it means to say one element is less than another. Once this decision has been made, the `compareTo` method is implemented in a way that reflects the decision.

NOTE *The class of the objects being sorted using the* `Collections` *class's* `sort` *method, must implement the interface* `Comparable` *or a translation error is generated.*

In this case, it was decided that one transcript is less than another if its GPA is lower, and the coding of the `compareTo` method on lines 21–27 of Figure 13.27 reflects that decision.

Consistent with the description of the `compareTo` method in the interface `Comparable`, the method returns a negative number on line 26 if the object that invoked it is less than the object passed to it, (otherwise it returns a positive number for greater than and zero for equal). The implements clause included in the heading of the class indicates that the `compareTo` method will be passed a reference to a `Transcripts` instance. Omitting the parameter list in the implements cause on line 1 of Figure 13.27 would result in a translation error on that line and line 30 of Figure 13.26.

Lines 38 and 39 of Figure 13.26 invoke the `Collections` class's `max` and `min` methods that return a reference to the maximum and minimum elements in the collection. These methods also invoke the `Transcripts` class's `compareTo` method to compare two elements, so they return a reference to the elements in the collection with the highest and lowest GPA (the third group of outputs in Figure 13.28).

Lines 42–48 of Figure 13.26 produce the fourth group of outputs shown in Figure 13.28. Before the output is performed, line 44 invokes the `Collections` class's `replaceAll` method. The method is passed the `LinkedList` collection object `underGrads` and the two `Transcript` objects `t1` and `t2`. This causes all occurrences of `t1` in the collection to be replaced with `t2` (in this case, just one replacement is made).

Line 52 uses the `Collections` class's `binarySearch` method to find and output the current location of object `t4`, which at this point is the second element (location 1 in the collection). Line 57 invokes the `swap` method to swap the first (location 0) and the last (location 3) elements in the collection. The collection is then output within the `while` loop that begins on line 59 that uses an iterator to traverse and output the list.

```
1    import java.util.LinkedList;
2    import java.util.Collections;
3    import java.util.List;
4    import java.util.ListIterator;
5
6    public class LinkedListApp
7    {
8      public static void main(String[] args)
9      {
10       LinkedList <Transcripts> underGrads = new LinkedList<Transcripts>();
11       Transcripts t1 = new Transcripts("Dana", 3.5, 45);
12       Transcripts t2 = new Transcripts("Carol", 3.8, 45);
13       Transcripts t3 = new Transcripts("Alice", 1.7, 22);
14       Transcripts t4 = new Transcripts("Bob", 2.6, 120);
15
16       underGrads.add(t1); //Add the transcripts to the list
17       underGrads.add(t2);
18       underGrads.add(t3);
19       underGrads.add(t4);
20
21       //Output the transcripts sequentially
22       System.out.println("\nAll transcripts in order of entry");
23       for(int i = 0; i < underGrads.size(); i++)
24       {
25         System.out.println(underGrads.get(i));
26       }
27
28       //The Collections class's sort method
29       System.out.println("\nAll transcripts in sorted order by GPA");
30       Collections.sort(underGrads);
31       for(int i = 0; i < underGrads.size(); i++)
32       {
33          System.out.println(underGrads.get(i));
34       }
35
36       //The Collections class's min and max methods
37       System.out.println("\nHighest GPA is " +
38                              Collections.max(underGrads));
```

```
    System.out.println("Lowest GPA is " + Collections.min(underGrads));

    //The Collection class's replaceAll method
    System.out.println("\nAll transcripts replacing "+
                       "Dana's transcript with Carol's transcript");
    Collections.replaceAll(underGrads, t1, t2);
    for(int i = 0; i < 4; i++)
    {
      System.out.println(underGrads.get(i));
    }

    //The Collections class's binarySearch method
    System.out.println("\nt4, Bob, is currently at location " +
                       Collections.binarySearch(underGrads, t4));

    //Use of an iterator
    System.out.println("\nAll transcripts output using an iterator " +
                       "after locations 0 and 3 were swapped");
    Collections.swap(underGrads,0, 3);
    ListIterator <Transcripts> anIterator = underGrads.listIterator(0);
    while (anIterator.hasNext())
    {
      System.out.println(anIterator.next());
    }
  }
}
```

Figure 13.26
The application **LinkedListApp**.

```
public class Transcripts implements Comparable <Transcripts>
{
  String name;
  double gpa;
  int creditsEarned;

  public Transcripts(String name, double gpa, int creditsEarned)
  {
    this.name = name;
    this.gpa = gpa;
    this.creditsEarned = creditsEarned;
  }

  public String toString()
  {
    return "name: " + name +
           "; gpa: " + gpa +
           "; credits earned: " + creditsEarned;
```

```
  }

  public int compareTo(Transcripts aTranscript)
  {
    //Defines the natural order of transcripts
    int gpa1 = (int) (gpa * 100);
    int gpa2 = (int) (aTranscript.gpa * 100);
    return gpa1 - gpa2;
  }
}
```

Figure 13.27
The class **`Transcripts`**.

```
All transcripts in order of entry
name: Dana; gpa: 3.5; credits earned: 45
name: Carol; gpa: 3.8; credits earned: 45
name: Alice; gpa: 1.7; credits earned: 22
name: Bob; gpa: 2.6; credits earned: 120

All transcripts in sorted order by GPA
name: Alice; gpa: 1.7; credits earned: 22
name: Bob; gpa: 2.6; credits earned: 120
name: Dana; gpa: 3.5; credits earned: 45
name: Carol; gpa: 3.8; credits earned: 45

Highest GPA is name: Carol; gpa: 3.8; credits earned: 45
Lowest GPA is name: Alice; gpa: 1.7; credits earned: 22

All transcripts replacing Dana’s transcript with Carol’s transcript
name: Alice; gpa: 1.7; credits earned: 22
name: Bob; gpa: 2.6; credits earned: 120
name: Carol; gpa: 3.8; credits earned: 45
name: Carol; gpa: 3.8; credits earned: 45

t4, Bob, is currently at location 1

All transcripts output using an iterator after locations 0 and 3 were swapped
name: Carol; gpa: 3.8; credits earned: 45
name: Bob; gpa: 2.6; credits earned: 120
name: Carol; gpa: 3.8; credits earned: 45
name: Alice; gpa: 1.7; credits earned: 22
```

Figure 13.28
The output produced by the application **`LinkedListApp`**.

Iterators

An iterator is an object that can be used to move through (traverse) an ordered list, usually in a forward (increasing element location) direction or in a backward (decreasing element location) direction. The iterator's class ordinarily contains methods that can fetch, add, and remove the list element just after or just before the iterator's current location and determine if there is an element just before and just after the iterator's current location. The API class `ListIterator` provides all of these methods.

An iterator's current location is always either between two elements of a list, just before the first element, or just after the last element. Figure 13.29 shows the five possible iterator positions for a collection that contains four elements. The `ListIterator` class's `hasPrevious` method returns `false` when the iterator is positioned before the first element in the list, and its `hasNext` method returns `false` when the iterator is positioned after the last element in the list. Otherwise, they return `true`. The method `next` returns the element to the iterator's right *and* then advances the iterator one position to the right. The method `previous` returns the element to the iterator's left and then advances the iterator one position to the left.

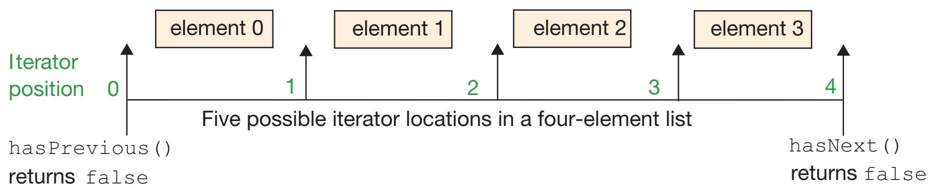


Figure 13.29
Positions of a list iterator.

Line 58 of Figure 13.26 invokes the `LinkedList` class's `listIterator` method, which creates and returns an instance of the class `ListIterator`. The method is passed the iterator position 0, which, as shown in Figure 13.29, positions the iterator before the first element in the list. The iterator object `anIterator`, declared and initialized on line 58, is used to invoke the `hasNext` method in the Boolean condition of the `while` loop that begins on line 59. The output statement on line 61 of the loop outputs all of the elements of the list by invoking the iterator's `next` method inside the `println` method, which returns the address of the next element of the array and advances the iterator.

The use of an iterator in a `while` loop is a more time-efficient way to traverse a list than the other output traversals that use `for` loops in the application `LinkedListApp` (e.g., lines 45-48). The reason it is more efficient is that the iterator maintains its position in the list after each iteration of the loop, which means it only has to advance one element during the next iteration. In contrast each iteration of the application's `for` loops that do not use an iterator begins at the first element of the list. As a result, the number of elements the `for` loops have to traverse to output the *last* element in the list is the same number of elements traversed by the `while` loop to output *all* of the elements in the list.

13.4.4 The `HashSet`, `TreeSet`, and `LinkedSet` Classes

The framework collection classes `HashSet`, `TreeSet`, and `LinkedSet` implement the interface `Set`. An instance of a class that implements `Set` cannot contain duplicate objects in its collection, as determined by the implementation of the `equals` method in the objects' class. For example if the elements in the collection were the Major League Baseball teams, an attempt to use the class's `add` method to include a duplicate team object in the collection would result in a non-operation, and the `add` method would return `false`.

These three classes do not contain a `get` method for fetching elements from the collection. They do have a method named `iterator` that can be used to attach an iterator to an instance of these classes, which is used in the following code fragment to fetch and output all of the elements of the `TreeSet` object `ts`. An instance of a `TreeSet` maintains its elements in a sorted order, and a traversal of it using an iterator returns the elements in ascending order. The following code fragment outputs *4 7 22* to the system console:

```
TreeSet <Integer> ts = new TreeSet<Integer>();

ts.add(22); //Autoboxing
ts.add(4);
ts.add(7);

Iterator anIterator = ts.iterator();
while (anIterator.hasNext())
{
  System.out.print(anIterator.next() + " ");
}
```

Instances of the `HashSet`, `TreeSet`, and `LinkedSet` classes expand to accommodate the number of elements added to the collection. Elements can be removed using the classes's `remove` method. Several of the `Collections` class's methods, such as `max`, `min`, `addAll`, and `disjoint` can be used to process elements in these three collection classes. The class `LinkedSet` extends `HashSet`.

13.4.5 The `ArrayDeque` and `PriorityQueue` Classes

The framework collection classes `ArrayDeque` and `PriorityQueue` implement the interfaces `Deque` and `Queue`, respectively. An instance of these classes can contain duplicate objects in its collection. In addition, the objects in the collection have a first-in-first-out ordering imposed upon them. As a result, the methods implemented in these classes to add and remove objects are not passed an integer index to specify the new element's location in the collection.

Instances of these classes would be good candidates for collecting the maintenance work orders of an apartment complex. Ordinarily work orders, such as polishing doorknobs, rats running around the kitchen, replacing a light, and toilets backing up, are added in the order in which they are received. Obviously, some of these are more urgent than others.

Each class adds its own embellishment to the first-in-first-out ordering of a traditional queue collection. The class `ArrayDeque` has a method named `add` that adds a new element to the end or rear of the collection, and a method named `remove` that removes an element from the front of the collection. When elements are added to the collection with the `addFirst` method and removed with the `remove` method, the collection object emulates a *last-in-first-out* collection (called a *stack*).

The `PriorityQueue` class's first-in-first-out ordering would be more accurately described as first in - with the highest priority - first out, in that the elements are maintained in a priority ordering. An element's priority is determined by the natural ordering of the objects as defined by their class's implementation of the `compareTo` method specified in the interface `Comparable`. The collection object uses the integer returned from the `compareTo` method as the element's priority. Normally, one of the element's data members is used to designate the element's priority and is then used to determine the integer returned from the implementation of the `compareTo` method. The element with the lowest natural ordering has the highest priority (e.g., a priority of 1 is higher than a priority of 2).

The application `Queues`, shown in Figure 13.30, demonstrates the use of the `ArrayDeque` and `PriorityQueue` classes to collect maintenance work orders that are objects in the class `WorkOrder` (Figure 13.31). An instance of a `WorkOrder` contains three data members: an apartment number, a description of the work to be performed, and a priority declared on lines 3–5 of the class. The output produced by the application is shown in Figure 13.32.

The application `Queues` (Figure 13.30) declares an instance of `ArrayDeque` named `tasks` on line 7, which specifies that all of the collection's elements will be `WorkOrder` instances. Lines 10–13 uses the class's `add` method to add four new work orders to the collection, which are then dequeued using the `remove` method and output (line 18) inside the `while` loop that begins on line 28. The loop's Boolean condition uses the `ArrayDeque`'s `size` method to determine when all of the work orders have been removed from the queue. This method returns the number of elements in the collection. As shown in the top of Figure 13.32, the work orders are fetched and output by lines 16–19 in the chronological order in which the work orders were added to the collection `tasks` on lines 10–13.

To remedy the fact that the use of the `tasks` collection has the maintenance man polishing a door knob while rats are running around the apartment complex and toilets are backing up, the `PriorityQueue` object `ptasks` is declared on line 8 of the application. The work orders are added to this collection on lines 23–26 in the same chronological order in which they were added to the `tasks` collection (lines 10–13). The work orders in the `ptasks` collection are fetched and output on line 32 inside a `while` loop (line 30) that duplicates the loop used to output the `tasks` collection (line 16).

This time, the output is prioritized by the value of the work orders' data member `priority`. The two highest priority work orders, given a priority of 1 on lines 24 and 26 when they were added to the collection, are output before the two lower priority work orders. The *Rats running around the kitchen* work order is placed in front of the *Toilet backing up* work order on the queue because it was added to the queue chronologically before the toilet maintenance request. Even though the

Polish doorknob work order was the first order added to the `ptasks` collection (line 23), it is fetched by line 32 after the two priority 1 work orders and the priority 7 work order within the collection because it has a lower priority: 10.

To define the natural ordering of `WorkOrder` objects, lines 20–23 of the `WorkOrder` class (Figure 13.31) implements the `compareTo` method. The integer returned from this method is used in the manner described in the `Comparable` interface by a `PriorityQueue` collection object to determine its elements' priority ordering. The method is passed a `WorkOrder` object as designated in the implements clause in the class's heading. The implements clause, which includes a type parameter list consistent with the method's parameter list on line 20, must be included in the class's heading.

```
1    import java.util.*;
2
3    public class Queues
4    {
5      public static void main(String[] args)
6      {
7        ArrayDeque <WorkOrder> tasks = new ArrayDeque<WorkOrder>();
8        PriorityQueue <WorkOrder> ptasks = new PriorityQueue<WorkOrder>();
9
10       tasks.add(new WorkOrder("1C", "Polish door knob.", 10));
11       tasks.add(new WorkOrder("8A", "Rats running around kitchen.", 1));
12       tasks.add(new WorkOrder("8A", "Replace light bulb in hall.", 7));
13       tasks.add(new WorkOrder("12B", "Toilet backing up.", 1));
14
15       System.out.println("Work Orders Non-prioritized by an ArrayQueue");
16       while(tasks.size() != 0)
17       {
18         System.out.println(tasks.remove());
19       }
20
21       System.out.println();
22
23       ptasks.add(new WorkOrder("1C", "Polish door knob.", 10));
24       ptasks.add(new WorkOrder("8A", "Rats running around kitchen.", 1));
25       ptasks.add(new WorkOrder("8A", "Replace light bulb in hall.", 7));
26       ptasks.add(new WorkOrder("12B", "Toilet backing up.", 1));
27
28       System.out.println("Work Orders Prioritized by " +
29                          "a PriorityQueue");
30       while(ptasks.size() != 0)
31       {
32         System.out.println(ptasks.remove());
33       }
34     }
35   }
```

Figure 13.30
The application **Queues**.

```java
public class WorkOrder implements Comparable <WorkOrder>
{
  String apartmentNumber;
  String description;
  int priority;

  public WorkOrder(String location, String description, int priority)
  {
    apartmentNumber = location;
    this.description = description;
    this.priority = priority;
  }

  public String toString()
  {
    return "Apartment " + apartmentNumber +
           ", " + description;
  }

  public int compareTo(WorkOrder aWorkOrder)
  {
    return priority - aWorkOrder.priority;
  }
}
```

Figure 13.31
The class **`WorkOrder`**.

```
Work Orders Non-prioritized by an ArrayQueue
Apartment 1C, Polish doorknob.
Apartment 8A, Rats running around kitchen.
Apartment 8A, Replace light bulb in hall.
Apartment 12B, Toilet backing up.

Work Orders Prioritized by a PriorityQueue
Apartment 8A, Rats running around kitchen.
Apartment 12B, Toilet backing up.
Apartment 8A, Replace light bulb in hall.
Apartment 1C, Polish doorknob.
```

Figure 13.32
The output produced from the application **`Queues`**.

13.4.6 The `HashMap`, `TreeMap`, and `LinkedHashMap` Classes

The framework collection classes `HashMap` and `LinkedHashMap` implement the interface `Map`, and the `TreeMap` class implements the `Map` and `SortedMap` interfaces. Instances of these classes store objects, called *values*, which are paired (associated) with another object called a *key*.

Each value object must be associated with a unique key object. The values stored in these collection classes are analogous to the elements stored in the collection classes previously discussed in this chapter. They are instances of *any* class.

When a key and a value pair are added to the collection using the class's `put` method, the key and the associated value are passed to the method. This establishes the key and value association. A value is fetched from the collection by invoking the class's `get` method and passing it the key associated with the value to be fetched. The following code fragment declares a `TreeMap` collection object named `patientInfo` whose keys are Strings and whose values are instances of the class `Patient` shown in Figure 13.33. It adds the object value `p1` and its associated string key *Jones* to the collection and then outputs the value after it is fetched from the collection by passing the key *Jones* to the `TreeMap` class's `get` method.

```
TreeMap<String, Patient> patientInfo = new TreeMap<String, Patient>();
 Patient p1 = new Patient("Tom Jones", "2/3/1989", "643 976-4545");

 //Save the key and value pair in the collection
 patientInfo.put("Jones", p1);

 //Fetch and output the value whose key is "Jones"
 System.out.println(patientInfo.get("Jones"));
```

The output produced by the code fragment's implicit invocation of the `Patient` class's `toString` method (lines 13–16 of Figure 13.33) is shown below:

name Tom Jones,   DOB: 2/3/1989,   Cell Number: 643 976-4545

Fetching a value from a collection by specifying a key associated with the value is the most commonly used mode of accessing values stored in a collection and is a feature supported by the API classes that implements the `Map` interface.

Instances of the classes `HashMap`, `TreeMap`, and `LinkedHashMap` expand beyond their initial default capacity to accommodate the number of values added to them. From a speed viewpoint, a `HashMap` collection object affords the best performance, followed closely by `TreeMap` instances. A `TreeMap` collection object imposes a sorted order on the values in the collection, and a `LinkedHashMap` collection object maintains the order in which the values were added to collection. A `HashMap` collection object imposes a pseudorandom order on the values. The class of keys associated with the values added to a `TreeMap` collection object must implement the interface `Comparable` to define the natural ordering of the keys stored in the collection object.

```
1   public class Patient
2   {
3     String name;
4     String DOB;
5     String cellNumber;
6
7     public Patient(String name, String DOB, String cellNumber)
8     {
```

```
    this.name = name;
    this.DOB = DOB;
    this.cellNumber = cellNumber;
  }
  public String toString()
  {
    return name + ", \tDOB: " + DOB + ", \tCell Number: " + cellNumber;
  }
}
```

Figure 13.33
The class **`Patient`**.

The application `TreeMapApp`, shown in Figure 13.34, demonstrates the use of a `TreeMap` collection object named `patientInfo` (declared on line 8) to store a collection of `Patient` object values and the techniques used to fetch values from the collection and output the entire collection. A set of inputs and the resulting outputs produced by the program are shown in shown in Figure 13.35. The class `Patient` is shown in Figure 13.33.

Lines 17–20 of Figure 13.34 add the objects declared on lines 10–13 to the collection object `patientInfo` by passing its `add` method a string key and an associated value. The keys are the last names of the patients. Although a value's key need not be contained in the value's object, in this case, the value object does contain the last as well as the first name of the patient.

The sentinel loop that begins on line 26 and ends on line 41 is used to repeatedly fetch and display a patient's information, given the patient's last name. The user is prompted to enter a person's last name before the loop begins (lines 23–25) and at the end of every loop iteration (lines 38–40) via an input dialog box (Figure 13.35a). If the user clicks Cancel in response to the prompt, the `showInputDialog` method returns a `null` value, and the loop ends.

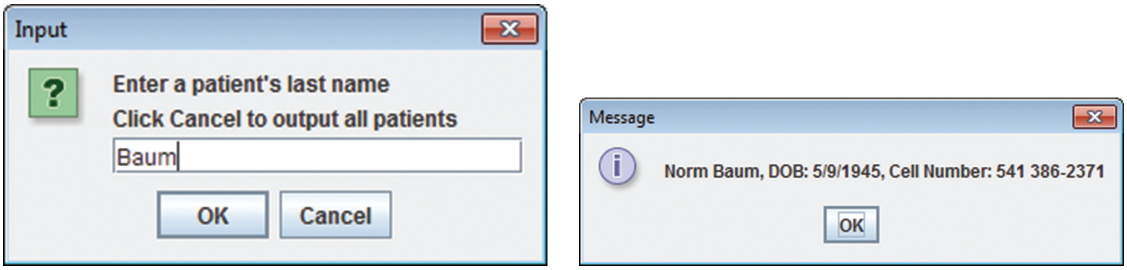
When the returned value is not `null`, the input string is passed to the collection object's `get` method on line 28. The `get` method returns the address of the value object in the collection associated with that string key or a `null` if the key is not associated with a value in the collection. When a `null` is returned, lines 31–32 inform the user that the person is not in the database (the collection). Otherwise, the person's information is output to a message dialog box on line 36 using an implicit invocation to the `Patient` class's `toString` method. A typical output is shown in Figure 13.35b.

When the `while` loop ends, the enhanced `for` loop that begins on line 44 fetches and outputs all of the value objects stored in the collection to the system console (shown in the bottom portion of Figure 13.35). The keys passed to the `get` method invoked on line 46 are sequentially accessed from the `Set` of keys returned from the collection object's `keySet` method invoked on the right side of line 44. The collection values are output in last-name sorted order because a `TreeMap` collection object maintains the key set in sorted order, and the set returned from the `ketSet` method reflects that ordering.

```java
import java.util.TreeMap;
import javax.swing.*;

public class TreeMapApp
{
  public static void main(String[] args)
  {
    TreeMap<String, Patient>patientInfo = new TreeMap<String, Patient>();
    String lastName;
    Patient p1 = new Patient("Tom Jones", "2/3/1989", "643 976-4545");
    Patient p2 = new Patient("Amy Adams", "8/5/1991", "643 531-2283");
    Patient p3 = new Patient("Norm Baum", "5/9/1945", "541 386-2371");
    Patient p4 = new Patient("Ray Rondo", "2/6/1998", "643 736-2949");
    Patient aPatient;

    //Save the key and value pairs in the collection
    patientInfo.put("Jones", p1);
    patientInfo.put("Adams", p2);
    patientInfo.put("Baum", p3);
    patientInfo.put("Rondo", p4);

    //Fetch and output a patient’s value (object)
    lastName = JOptionPane.showInputDialog("Enter a patient's last " +
                                           "name \nClick Cancel " +
                                           "to output all patients");
    while(lastName != null) //not a Cancel click
    {
      aPatient = patientInfo.get(lastName);
      if(aPatient == null) //key is not in collection
      {
        JOptionPane.showMessageDialog(null, "That person is not in" +
                                            "our data base");
      }
      else //output the value
      {
        JOptionPane.showMessageDialog(null, aPatient);
      }
      lastName = JOptionPane.showInputDialog("Enter a patient's last " +
                                             "name \nClick Cancel " +
                                             "to output all patients");
    }

    //Output all patients
    for (String key: patientInfo.keySet()) //all keys in the collection
```

```
45        {
46          aPatient = patientInfo.get(key);
47          System.out.println(aPatient);
48        }
49      }
50  }
```

Figure 13.34
The application **TreeMapApp**.



(a) (b)

Console Output:
Amy Adams, DOB: 8/5/1991, Cell Number: 643 531-2283
Norm Baum, DOB: 5/9/1945, Cell Number: 541 386-2371
Tom Jones, DOB: 2/3/1989, Cell Number: 643 976-4545
Ray Rondo, DOB: 2/6/1998, Cell Number: 643 736-2949

Figure 13.35
An input and the outputs produced by the application **TreeMapApp**.

The following code fragment produces the same output as the enhanced `for` loop on lines 44–48 of Figure 13.34 when substituted for it. It uses an `Iterator` object to traverse the key set.

```
// Use of an iterator to traverse a TreeMap instance (i.e, patientInfo)
Set <String> keys = patientInfo.keySet();
Iterator anIterator = keys.iterator();

while (anIterator.hasNext())
{
  String k = (String) anIterator.next();
  System.out.println(patientInfo.get(k));
}
```

13.5 STREAMS AND FUNCTIONAL PROGRAMMING

Streams and Lambda expressions were added to Java in Version 8. These enhancements effectively added a fourth programming paradigm, functional programming, to the three programming models Java already supported: procedural, object oriented, and generic programming. Within the functional model, the programmer describes what to do by sequentially invoking pre-coded functional methods, methods that return a value, rather than describing how to do it by coding the processes' algorithms.

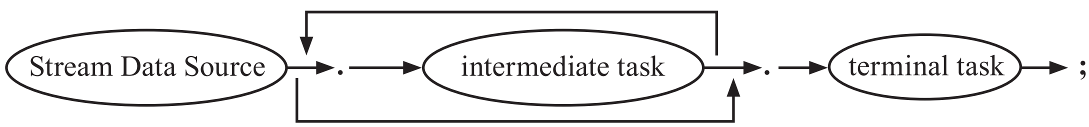
While this approach to programming has its disadvantages, a major advantage is that the functional methods implement operations commonly used to process data contained in collections and disk files, saving the programmer the time required to develop these methods. In addition, the streams that invoke them can be executed concurrently without the programmer having to write the multithreaded code. The functional paradigm also eliminates inadvertent interaction, referred to as *coupling*, between the modules that make up a program. For example, a sequence of code within this paradigm could not inadvertently change a class level variable.

A *Stream* is a programming construct that generates a sequence of objects, referred to as *data elements* which are then processed by a pipeline of one or more sequentially executed functional methods, referred to as intermediate *tasks* or intermediate *operations*. After a data element has been processed by an *intermediate* task, it is replaced with the task's returned object which is then processed by the next task in the pipeline. Data element n does not enter the pipeline until element n-1 has been processed by the last task in the pipeline, or element n-1 was removed from the pipeline and not replaced, or element n-1 reaches an intermediate task that operates on more than one element. The last task in the pipeline is referred to as the *terminal* task, and it produces the pipeline's result. A pipeline with two intermediate tasks is illustrated below.

Stream Data Source → .task1'sName → .task2'sName → .terminalTask'sName

As shown in the below syntax diagram, the coding of a stream begins with a stream source that generates the data elements. This is followed by zero or more intermediate task names each preceded by a dot, followed by one terminal task preceded by a dot and followed by a semicolon.

Syntax Diagram of a Stream



The following is the coding of a stream that conforms to this syntax, and produces the average of the numbers on line 1 that are less than or equal to 3.05.

```
1  DoubleStream.of(2.5, 3.9, 2.1, 3.1, 2.3) // data source
2               .filter(x -> x <= 3.05) // intermediate task
3               .average(); // terminal task
```

Line 1 is the stream's data source, line 2 is an intermediate task, and line 3 is a terminal task.

13.5.1 A Stream Data Source

The Stream Data Source portion of the stream's pipeline is analogous to a sequential loop whose code block is the stream's tasks coded in task number order. On the i^{th} iteration of the loop, the data element i of the stream's data source enters the loop, and is processed by the stream's tasks as previously described.

The data elements of the data stream can be a sequence of non-random or random numeric values, the elements of an array, the elements of a collection, or the contents of a file. The Interfaces `IntStream`, `LongStream`, and `DoubleStream` contain static methods to generate streams of non-random numbers. For example, line 1 of the below code produces a stream data source whose data elements are the integers 3 though 6 inclusive, and line 2 produces the equivalent sequence of doubles. Line 3 is an alternative way of coding line 1, however the `of` method is normally used to generate a non-sequential integer stream.

```
// Stream data sources that produce streams of numbers
1  IntStream.rangeClosed(3, 6)
2  DoubleStream.of(3.0, 4.0, 5.0, 6.0)
3  IntStream.of(3, 4, 5, 6)
```

To use elements of an array as the stream's data source, the `Arrays` class's static method stream is invoked and passed the array. The method returns a stream data source containing the elements of the array in ascending array index order. The array can be an array of numeric primitives or objects. Lines 1–2 below produce a stream whose elements are the five doubles coded on Line 1. Lines 5 and 8 produce streams whose elements are the instances of the class `String` and `Color` stored in the arrays created on lines 4 and 7 respectively.

```
// Streams of array elements
1 double[] doubles = {1.1, 5.2, 3.3, 16.4, 12.0};
2 Arrays.stream(doubles)
3
4 String[] StringObjects = {"Element1", "Element2", "Element3"};
5 Arrays.stream(StringObjects)
6
7 Color[] ColorObjects = {Color.RED, Color.GREEN, Color.BLUE};
8 Arrays.stream(ColorObjects)
```

To use the objects stored in a collection as a stream's data source the `Collection` interface's stream method is invoked on the collection instance. The below line of code creates a stream data source whose elements are the four instances of the `Transcript` class (Figure 13.27) created and added to the `LinkedList` collection named `undergrads` on lines 10–19 of Figure 13.26.

```
// A stream of a collection elements
underGrads.stream()
```

13.5.2 Stream Terminal Tasks

Table 13.2 presents a description of the 13 terminal tasks defined in the `Stream` interface. Some of these tasks are overloaded within this interface. The interfaces `IntStream`, `LongStream`, and `DoubleStream` contain equivalent tasks that operate on numeric streams, and include additional tasks that operate on numeric stream elements such as `average` and `sum`.

The parameter types of the methods in the first eight rows of Table 13.2 are functional interfaces, and therefore the argument passed to them can be a Lambda expression. These Lambda expressions would be implementations of the functional interface `Comparator` for the `min` and `max` tasks, the functional interface `BinaryOperator` for the `reduce` task, the functional interface `Predicate` for the `anyMatch` and `allMatch` and `noneMatch` tasks, and functional interface `Consumer` for the `for` and `forEachOrdered` tasks.

Table 13.2
The Terminal Tasks Defined in the Interfaces Stream

Terminal Task	Description
`Optional<T> min(Comparator<? super T> comparator)`	Returns an `Optional` object containing the minimum element in a stream determined by using the `comparator` passed to it
`Optional<T> max(Comparator<? super T> comparator)`	Returns an `Optional` object containing the maximum element in a stream determined by using the `Comparator` passed to it
`Optional<T> reduce(BinaryOperator<T> accumulator)`	Produces one element from the stream's elements using the `accumulator` passed to it
`boolean allMatch(Predicate<? super T> predicate)`	Determines if all the elements of a stream match the Boolean valued `predicate` passed to it
`boolean anyMatch(Predicate<? super T> predicate)`	Determines if any of the elements of a stream match the Boolean valued `predicate` passed to it
`boolean noneMatch(Predicate<? super T> predicate)`	Determines if none of the elements of a stream match the Boolean valued `predicate` passed to it
`void forEach(Consumer<? super T> action)`	Sequentially iterates through a stream's elements performing the `action` on each element
`void forEachOrdered(Consumer<? super T> action)`	Iterates through a stream's elements performing the `action` on each element in its defined encounter order
`long count()`	Returns the number of elements in the stream that reach this terminal task
`Object[] toArray()`	Creates and returns an array containing the stream's elements
`Optional<T> findFirst()`	Returns an `Optional` object containing the first element in the stream
`Optional<T> findAny()`	Returns an `Optional` object containing one of the stream's elements
`<R,A> R collect(Collector<? super T,A,R> collector)`	Places the stream's elements into one entity defined by `collector`

The `reduce` Terminal Task, `BinaryOperators` Interface, and the `get` Method

The `reduce` task repeatedly evaluates the binary operation within the Lambda expression it receives passing the expression a sequential pair of stream elements, beginning with the first two elements that reach the task. In this case the binary operation must be an operation performed on two stream elements of the same type, and it must return an element of that type. The result of the binary operation replaces the pair of elements in the stream, thus *reducing* the stream by one element. The returned value is then paired with the next stream element and they are processed by the Lambda expression and replaced with the returned result. This process continues until there is only one pair of elements in the stream. The result of processing that last pair of elements is the value retuned from the reduce task.

The following section of code produces the output shown below it. Line 2 is the stream's data source. Its elements are the four strings within the array defined on Line 1. The binary operator `x + y` in the Lambda expression passed to the `reduce` task on line 3, repeatedly generates the concatenation of two sequential strings that reach it, which then replaces them in the stream. After the third concatenation the stream is reduced to one string element which is wrapped in the `Optional` object reduce returns. The invocation of the `Optional` class's `get` method, at the end of the Lambda expression on line 3, extracts the string from the Optional object which is assigned to the variable `s` on line 2. Line 4 outputs the string `s`.

```
1    String[] StringObjects = {"Element1", "Element2", " ... ", "ElementN"};
2    String s = Arrays.stream(StringObjects)
3                      .reduce((x, y) -> x + y).get());
4    System.out.println(s);
```

The output string produced
Element1Element2 ... ElementN

When the `get` method is invoked on an object that does not contain a value, the method throws a `NoSuchElementException`. The class contains two methods, `ifPresent` and `isPresent`, to determine if the Optional object contains a value.

The numeric stream classes contain an overloaded two parameter version of the `reduce` task, whose first parameter becomes the (new) first element of the stream, and then the reduction process is performed using the Lambda expression passed to the tasks's second parameter. The following code segment outputs the value 22.2 calculated as: 2.2 + 1.0*2.0 + 2.0*2.0 + 3.0*2.0 + 4.0*2.0 = 22.2

```
double[] values = {1.0, 2.0, 3.0, 4.0};
double avg1 =
DoubleStream.of(values)
                .reduce(2.2, (x, y)-> x + y * 2.0); //2.0 becomes element 1
System.out.println("reduced1 " + avg1);
```

The `min` and `max` Tasks and the `Comparator` Interface

The `min` and `max` terminal tasks return the minimum and maximum value of the stream elements that reach the terminal task. To do this, they use the two parameter `Comparator` Lambda expression passed to them that defines the natural ordering of two stream elements as defined in the `Comparable` interface. The returned element is wrapped in an `Optional` object.

The following code uses the `max` task on line 3 to determine the `Transcript` class (Figure 13.27) object with the highest `gpa`, and then outputs that transcript. The stream's data source on line 2, is the `LinkedList` collection of four transcripts named `undergrads` created on lines 10-19 of Figure 13.26. The `compareTo` method invoked on a `Transcript` stream element within the Lambda expression on Line 3, is coded within the `Transcript` class on lines 21-27. The entire stream is coded within the argument passed to the `println` method invocation that begins on line 1. This is an alternative coding style when the stream's returned value is not going to be used beyond the sequence of code that contains the stream.

```
1   System.out.println("\nThe student with highest GPA is: " +
2   underGrads.stream()
3             .max((x, y) -> x.compareTo(y)).get() );
```

The output produced

The student with highest GPA is: name: Carol; gpa: 3.8; credits earned: 45

The `anyMatch`, `noneMatch`, `allMatch` Tasks and the `Predicate` Interface

The `Predicate` functional interface defines a one parameter method that returns a Boolean value. The `anyMatch`, `noneMatch`, and `allMatch` terminal tasks return `true` if the one parameter Lambda expression Predicate passed to them returns `true` for any stream element, no element, or all of the elements respectively. Otherwise, these terminal tasks return `false`. For example, the `anyMatch` terminal task on line 4 of the below code returns `true` because the stream of names defined in the array on line 1 contains the name Alice. If the `noneMatch` or `allMatch` tasks were used on line 3, they would return `false`.

```
1   String[] studentNames ={"Dana", "Carol", "Alice", "Bob"};
2   System.out.println("There is a student named Alice? " +
3         Arrays.stream(studentNames)
4               .anyMatch(x -> x.equals("Alice")) );
```

The output produced

There is a student named Alice? true

As shown on line 4 above, a `Predicate` Lambda expression contains one parameter to receive the stream object, and returns a Boolean value. If the stream objects are numeric values, the Lambda expression's code can simply be a Boolean expression containing relational and logic operators as shown below on line 4. The output produced by line 5 is given below it.

```
double[] gpas = {3.2, 3.6, 2.8, 4.0, 2.7};
boolean inRange =
DoubleStream.of(gpas)
            .allMatch(x-> x >= 2.0 && x <= 4.0);
System.out.println("All GPAs are in the range 2.0 to 4.0? " + inRange);
```

The output produced
All GPAs are in the range 2.0 to 4.0? true

The `forEach` and `forEachOrdered` Task, and the `Consumer` Interface

The `Consumer` functional interface defines a method that is intended to perform an action on the object passed to it. The method does not return a value, and so it is said to consume the object passed to it. The terminal tasks `forEach` and `forEachOrdered` are used to perform the action, defined in the Lambda expression passed to them, on each element of the stream that reaches them. The Lambda expression `x -> System.out.print(x)` is often used within the `forEach` and `forEachOrdered` methods to perform an output action on all of the elements that reach these terminal tasks, as shown on line 3 below.

```
1   String[] RGandBStringObjects = {"RED ", "GREEN ", "BLUE "};
2   Arrays.stream(RGandBStringObjects)
3         .forEach(x -> System.out.print(x));
```

The output produced by the above code
RED GREEN BLUE

An alternative coding of the Lambda expression that uses the method reference operator `::` incorporated into Java in Version 8 is `System.out::println`. This syntax can be used when a Lambda expression consists of one statement that invokes another method. When the translator encounters the method operator it considers the item on its right to be a method that is being invoked on the object on its left. To invoke a static method, the class name is coded on the left side of the operator. Within the context in which it is coded, the translator can deduce the Lambda Expression's parameters, and replaces it with the equivalent Lambda expression.

The following code uses the method reference operator on line 3 to output the four instances of the `Transcripts` class (Figure 13.27) contained in the `LinkedList` collection of transcripts, named `undergrads`, created on lines 10-19 of Figure 13.26. The `println` method invokes the `toString` method coded in the `Transcripts` class.

```
1   System.out.println("The students are:");
2   underGrads.stream()
3             .forEach(System.out::println);
```

The output produced by the above code
The students are:

name: Dana; gpa: 3.5; credits earned: 45
name: Carol; gpa: 3.8; credits earned: 45
name: Alice; gpa: 1.7; credits earned: 22
name: Bob; gpa: 2.6; credits earned: 120

The `forEach` and the `forEachOrdered` terminal tasks process the stream elements in the same order unless the stream is generated with the parallel or `parallelStream` method in the `BaseStream` Interface. When this is the case, the `forEachOrdered` task maintains the ordering of the stream elements and the `forEach` task does not.

The `toArray` and `collect` Terminal Tasks, and the `Collectors` Class

The `toArray` and the collect terminal tasks combine the elements of the stream into one entity. As its name implies, when the `toArray` task is used the entity returned is an array containing the stream's elements. The below sequence of code uses the `toArray` terminal task on line 3 to return an array of doubles containing the stream elements produced on line 2.

```
1   double[] arrayOfElements =
2   DoubleStream.of( 2.5, 3.9, 2.1, 3.1, 2.3)
3               .toArray();
4   System.out.println(Arrays.toString(arrayofElements));
```

The output produced
[2.5, 3.9, 2.1, 3.1, 2.3]

When the `collect` task is used the entity returned is normally a collection of objects such as a `Map`, a `List` or a `Set`, however it could also be a non-collection entity such as a string or a numeric. The returned entity depends on the static method invocation passed to the task, which is usually a static method defined in the `Collectors` class that returns a reference to `Collector`. A detailed discussion of these methods is beyond the scope of this book.

13.5.3 Stream Intermediate Tasks

Table 13.3 presents a description of the eight intermediate tasks defined in the `Stream` interface. The interfaces `IntStream`, `LongStream`, and `DoubleStream` contain equivalent tasks that operate on numeric streams, and include additional tasks that operate on numeric stream elements, such as `average`, and tasks to convert one type of numeric stream to another type of numeric stream.

The parameter type of the methods in the first four rows of Table 13.3 are functional interfaces, and therefore the argument passed to them can be a Lambda expressions. These Lambda expressions would be implementations of the functional interface `Predicate` for the `filter` tasks, the functional interface `Function` for the `map` and `flatMap` tasks, and functional interface `Consumer` for the for the `peek` task.

Table 13.3
Intermediate Tasks

Intermediate Task	Description
`Stream <T> filter(Predicate<? super T> predicate)`	Returns a `Stream` whose elements are this stream's elements for which the `predicate` (Boolean-valued function) evaluates to `true`
`Stream <T> map(Function<? super T,? extends R> mapper)`	Returns a Stream consisting of the results of applying the given `mapper` function to the elements of this stream.
`Stream <T> peek(Consumer<? super T> action)`	Returns a stream consisting of the elements of this stream, additionally performing the `action` on each element as elements are consumed from the resulting stream.
`Stream <T> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`	Returns a Stream whose elements are the *streams* produced by applying the `mapper` function to each element of the stream
`Stream <T> sorted()`	Returns a `Stream` whose elements are this stream's elements sorted in natural order
`Stream <T> skip(long n)`	Returns a `Stream` whose elements are this stream's elements with the first n elements eliminated from it
`Stream <T> limit(long maxSize)`	Returns a `Stream` whose elements are this stream's first `maxSize` elements that exist
`Stream <T> distinct()`	Returns a `Stream` whose elements are this stream's distinct elements such that no two elements reference the same object

The `filter` and `sorted` Intermediate Tasks, and the `Function` Interface

The `filter` intermediate task eliminates an element from the stream passed to its one parameter Lambda expression `predicate` if the Lambda expression's Boolean condition evaluates to `false`. The task is said to "filter" these elements out of the stream. The `sorted` intermediate task returns a stream containing the stream's elements in their natural sorted order. To do this it invokes the stream element's `compareTo` method within its sorting algorithm.

Line of 2 of the below code creates a stream data source whose elements are the four instances of the `Transcript` class (Figure 13.27) created and added to the `LinkedList` collection named `undergrads` on lines 10–19 of Figure 13.26. The `filter` task on line 3 returns a stream that includes only those transcript elements whose `gpa` is 2.0 or higher. Line 4 sorts these elements in GPA ascending order using the Transcripts class's `compareTo` method coded on lines 21–27 of Figure 13.27. The output produced by lines 1 and 5 is shown below the code.

```
System.out.println("The students with GPA >= 2.0 in sorted GPA order");
   underGrads.stream()
            .filter(x -> x.gpa >= 2.0) //gpa has package access
            .sorted() //invokes compareTo method in Transcripts class
            .forEach(x -> System.out.println(x));
```

The output produced by the above code sequence
The students with GPA >= 2.0 in sorted GPA order
name: Bob; gpa: 2.6; credits earned: 120
name: Dana; gpa: 3.5; credits earned: 45
name: Carol; gpa: 3.8; credits earned: 45

The `flatMap`, `map`, `distinct`, and `peek` Intermediate Tasks

Consider the six line code segment shown in Figure 13.36. As we have learned, line 1 generates a stream whose elements are 1, 2, and 3 and line 6 outputs the entire stream of elements that reaches it. If lines 2–5 were eliminated from it, the output would be the first line of output shown below the code segment. The functionality of the `flatMap`, `map`, `distinct`, and `peek` intermediate tasks on lines 2, 3, 4 and 5 will be illustrated by progressively adding each of these lines in between lines 1 and 6, executing each resulting code segment, and examining the resulting output. This will produce the last for outputs at the bottom of Figure 13.36.

```
1  IntStream.rangeClosed(1, 3)
2           .flatMap(x -> IntStream.rangeClosed(1, x))
3           .map(x -> x * 10)
4           .distinct()
5           .peek(x -> System.out.print(x + "  "))
6           .forEach((x)-> System.out.print(x + "  "));
```

Output produced by including
Lines 1 and 6: 1 2 3
Lines 1–2 and 6: 1 1 2 1 2 3
Lines 1–3 and 6: 10 10 20 10 20 30
Lines 1–4 and 6: 10 20 30
Lines 1–5 and 6: 10 10 20 20 30 30

Figure 13.36
The use and functionality of the intermediate tasks `flatMap`, `map`, `distinct` and `peek`.

Including line 2 of Figure 13.36 in between lines 1 and 6 incorporates the `flatMap` intermediate task into the stream. This expanded three line code segment changes the output to the second line of output shown at the bottom of the figure. When an element enters the `flatMap` task, it is mapped into (replaced with) the stream returned from `flapMap` that consists of zero or more elements produced by the Lambda expression passed to its `Function` parameter.

Line 2's Lambda expression invokes the `IntStream` class's static method `rangeClosed`, passing its parameter the integer 1 and the stream's integer element. This method returns a stream whose elements are the inclusive integers between the two arguments passed to it. The three elements 1, 2 and 3 generated by line 1 are therefore replaced with the three streams 1, 1 2, and 1 2 3 respectively. These 6 elements are then output on line 6.

Including line 3 in between lines 1–2 and 6 of Figure 13.36 incorporates the map intermediate task into the stream. This expanded four line code segment changes the code segment's output to the third line of output in the lower portion of the figure. When an element enters the `map` task, it is mapped into (replaced with) the value returned from the Lambda expression passed to its `Function` parameter.

Line 3's Lambda expression simply multiplies the element passed to it by 10 and returns the value which replaces the element in the stream. The six elements 1, 1, 2, 1, 2, 3 generated by line 2 are therefore replaced with the six stream elements 10, 10, 20, 10, 20, and 30. These 6 elements are then output on line 6.

Including line 4 in between lines 1–3 and 6 of the above code incorporates the `distinct` intermediate task into the stream. This expanded five line code segment changes the code segment's output to the fourth line of output in the lower portion of the figure. When an element enters the `distinct` task, it is eliminated from the stream if the task previously processed an element with the same value. And so, the six stream elements produced by line 3 10, 10, 20, 10, 20, and 30 are reduced to three distinct elements 10, 20, and 30. These 3 elements are then output on line 6.

Completing the code segment in Figure 13.36 by adding line 5 in between lines 1–4 and 6 incorporates the `peek` intermediate task into the stream. This expanded six line code segment changes the code segment's output to the fifth line of output at the bottom of the figure. When an element enters the `peek` task, it is consumed by the task, but not eliminated from the stream (as its name peek implies), because the task returns a stream with the element replaced by itself. In addition, the element and can be operated on by the Lambda expression passed to the task's `Consumer` parameter.

The operation performed on line 5 simply outputs an element followed by two spaces. Then line 6 outputs the element's replacement followed by two spaces. And so each one of the stream elements that enters the `peek` operation is output twice, once on line 5 and once on line 6, before the next element enters the stream. The resulting output is the last line of output shown at the bottom of Figure 13.36, which is produced when all six lines of the code segment in the figure are executed.

The `limit` and `skip` Intermediate Tasks

The `limit` and `skip` intermediate tasks both reduce the size of the stream. The `limit` task returns a stream consisting of the stream's first n elements, where n is the `long` integer passed to it. The `skip` task returns a stream with the first n elements removed from it, where n is the `long` integer passed to it. For example, if the stream elements are 10, 20, and 30, `.limit(2)` would replace the stream with a stream whose elements are 10 and 20 and `.skip(2)` would replace the stream with a stream containing the element 30.

13.6 CHAPTER SUMMARY

Generic implementations of methods and classes extend their reusability, thereby reducing the time and cost required to develop a software product. A generic method is an implementation of an algorithm that can process any type object passed to one or more of its parameters. Within the method's parameter list, the names of the parameters are preceded by a generic type placeholder, such as `T` or `T1`, instead of the name of a specific class. When a primitive value is passed to a parameter that uses a generic placeholder, it is autoboxed, and the address of the wrapper object is passed to the method.

All of the placeholders used in the method's parameter list must appear in a generic parameter list included in the method's signature just before its returned type: e.g., `<T, T1>`. One of these placeholders can also take the place of a specific returned type within the method's signature, and the placeholders can be used within the method to declare reference variables that can reference the type of objects passed the method's generic parameters. When a wrapper class object is returned from a generic method, it is unboxed before it is assigned to a primitive value. The `copy` method in the `Arrays`' class or an instance of an `ArrayList` is used to copy generic arrays passed to a method because Java does not support the declaration of a generic array.

Using the syntax of generics, one method can be written that produces an annotated output of an array of any type of object passed to it, but in this case, the class of the objects contained in the array would have to include a `toString` method. The class of the objects passed to a generic method must indicate that it implements an interface that defines the signature of the methods invoked within the generic method to operate on the objects passed to it. Typically, they are API interfaces such as `Comparable` and `Cloneable`. Like non-generic methods, generic methods can be overridden and overloaded. The version of an invoked overloaded method is identified using a best-fit protocol involving the argument and the method's parameter types.

Generic classes can contain generic methods, and generic placeholders can be used to specify the type of one or more of the class's data members. All of the placeholders used in the class must appear in a generic parameter list that is coded in the class's heading just after its class name. To maximize the type checking performed by the translator, the declaration of an instance of a generic class should include a generic parameter list that specifies the type of each of the class's placeholders for the object being declared. If the class implements an interface whose method signatures are generic, the class names passed to the type parameters of the class's implementation of the method should be included in its implements clause, for example:

```
className implements Comparable<theArgumentsClassName>.
```

The API collections framework contains a group of generic classes that implement many of the classic data structures used to efficiently store and process large data sets. These implementations include the classes `LinkedList`, `ArrayList`, `ArrayDeque`, `PriorityQueue`, and `ArrayBlockingQueue`, whose items are sequentially accessed, and the classes `HashMap`, `TreeMap`, and `LinkedHashMap`, whose values are accessed by specifying an object, called a key, that is associated with an object when it is added to the data set. The `LinkedList` and `LinkedHashMap` classes provide the poorest performance from a speed viewpoint. The collections framework also includes several interfaces used by its generic classes and a class named `Collections` whose methods can perform common operations on the items and values stored in instances of its collection classes such as sorting, searching, swapping, and locating maximums and minimums.

Knowledge Exercises

1. True or False:
   a) Objects can be passed to parameters that use generic placeholders.
   b) Primitive values can be passed to generic parameters.
   c) The parameter lists of generic methods can include parameters of primitive types and/or class names.
   d) The placeholder used as a method's returned type must appear in the method's type parameter list.
   e) A type parameter list can include type placeholders that are not used in the method's parameter list.
   f) A generic method can declare a reference variable using one of its generic type placeholders.
   g) An array of any type of object can be passed to a parameter of a generic method.
   h) A generic method can declare an array of one of its generic types.
   i) Generic methods can be overloaded.
   j) Generic methods can be overridden.
   k) The syntax of the invocation statement used to invoke a non-generic method is the same syntax used to invoke a generic method.
2. Give the signature of a generic method:
   a) Named `output` that has two generic parameters and does not return a value
   b) Named `find` that has one generic parameter and two integer parameters and returns an integer
   c) Named `clone` that returns an instance of the object passed to its generic parameter
   d) Named `min` that is passed an array of any type of object and returns a reference to one of the array's elements
3. Give the statement(s) to make a copy of an array passed to a parameter named `values` whose generic placeholder is `T1`.
4. Give the signature of a generic method with one parameter that operates on the object passed to its parameter using a method with no parameters defined in the interface `Addable`.
5. True or False:
   a) Generic classes can contain generic methods.
   b) Generic classes can contain non-generic methods.
   c) The heading of a generic class must contain a generic parameter list.
   d) The type specification of a generic class's data members can be a generic placeholder.
   e) Generic classes can extend other classes.
   f) Generic classes can implement interfaces.

6. Give the heading of a generic class named `g6Class` whose code uses two generic placeholders.

7. Give the heading of a generic class named `g7Class` that implements the `compareTo` method defined by the interface `Comparable` to compare the object that invoked it to another `g7Class` object.

8. Give a type-safe declaration of an instance of the class `ArrayList` that will be used to store 200 `String` objects.

9. Give a type-safe declaration of a reference variable that can reference the object declared in Exercise 8.

10. True or False:
    a) The collections framework is part of the API.
    b) The collections framework contains many generic classes that implement data structures used to efficiently store large data sets.
    c) The items stored in an instance of the framework class `LinkedList` can be associated with a key.
    d) The `sort` method in the `Collections` class can be used to sort the values of an instance of the framework class `HashMap`.
    e) The values stored in an instance of the framework class `TreeMap` are maintained in sorted order based on the keys with which they are associated.
    f) An iterator can be used to sequentially traverse through the elements stored in the framework collection classes.

11. Give the name of a frameworks class you could use to store a data set whose elements would be fetched by specifying a value associated with a string such as *Mary Smith*.

12. Give the name of a frameworks class you could use to store a data set whose elements would be fetched on a first-in-first-out basis.

13. Give a type-safe declaration statement to declare an object in the class identified in Exercise 12 that will store values that are objects in the class `PhoneListing`.

Programming Exercises

1. Write a generic method that outputs the three objects passed to it to the system console, each on a separate line. Test it by passing it a `String`, `Double`, and `Integer` object. (Why are these three classes used in this programming exercise?)

2. Write an overloaded version of the method described in Programming Exercise 1 that adds an integer primitive parameter to the method's signature and outputs it, too. Test it by passing the method an `Integer`, `Double`, and `String` object and an integer primitive value.

3. Write a generic method that outputs the first of two different objects passed to it to the system console and returns the second object passed to it. Test it by passing it a `String` and an `Integer` object and then outputting the returned object.

4. Write a generic method that returns the larger of two objects of the same type passed to it. Test it by passing it two `String` objects and two `Integer` objects. Then output the returned objects to the system console. (Why are the `String` and `Integer` classes used in this programming exercise?)

5. Write a generic method that sorts and outputs an array of objects passed to it to the system console, each on a separate line. Test it by passing it an array of `Integer` objects.

6. Expand the method described in Programming Exercise 5 so it also makes a copy of the sorted array and returns it. Test your method by outputting the returned array.

7. Write a generic data structure class named `arrayDS` that uses an `ArrayList` object to store any type of object passed to its `insert` method and returns the object whose index is passed to its `fetch` method. Test the class by declaring a type-safe instance of the class that can store a set of salaries: `Double` objects. Then, accept a given number of salaries and insert them into the data structure. When the input is complete, ask the user for the item number to fetch from the data structure and output it to the system console.

8. Expand Programming Exercise 7 to include two additional lines of output that contains the maximum and minimum salaries in the data structure. Use the methods in the framework `Collections` class to determine the salaries to be output.

9. Declare a type-safe data structure instance of the framework class `HashMap` whose keys are `String` objects (a person's name) and whose value is a `Double` object (the person's weight). Then accept a given number of persons' names and weights and insert them into the data structure. Fetch back the weight of the person whose name is input by the user, and output it to the system console. Finally, use an iterator to output the names and weights of all the objects in the data structure, one person per line.

Enrichment

Investigate the new features of Java 8, especially the use of lambda expressions and generics.

Investigate the differences between a deque and a queue.

Oracle's Processing Data with Java SE 8 Streams, Part 1

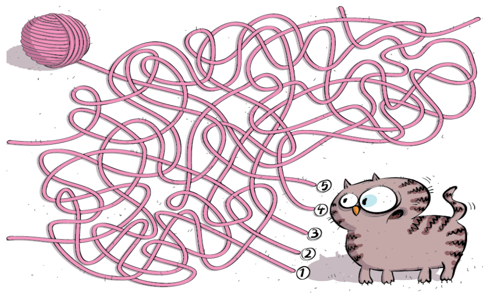
https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html

References

Horstmann, Cay. *Java SE 8 for the Really Impatient*. Upper Saddle River, NJ: Pearson, 2014.

CHAPTER 14

MULTITHREADING AND CONCURRENCY



In this chapter

In this chapter, we will learn the techniques used to divide a program into two or more independent execution paths, called *threads*, and how to share information among them them. Using these techniques, one of a program's threads can perform a time-consuming calculation on a set of inputs while another thread is accepting the next set of inputs. We will learn that modern operating systems can give the impression that threads are in concurrent execution or actually execute them concurrently.

We will discuss the problems associated with threads sharing data, the techniques and Java constructs used to avoid these problems, and the meaning of the term *thread safe* used in the description and implementation of several Java API classes such as the generic collection class `ArrayBlockingQueue`. We will learn how to write methods and classes that are thread safe using Java's synchronized method construct and its `synchronized` statement.

The Java syntax used to implement thread classes, create thread objects, and initiate and terminate their execution will be presented. We will learn that threads have a lifetime, and that during this lifetime, a Java thread exists in one of six states. The restrictions imposed on a thread while in these states, and the events that cause threads to transition from one state to another, will be discussed. During this discussion, we will be introduced to methods such as `wait`, `sleep`, and `notify`, which a programmer can use to transition a thread from one state to another.

After successfully completing this chapter, you should:

- Understand what threads are, how they are used, and how they can share information
- Know how to divide a program into one or more threads or execution paths

- Understand the six transition states of threads
- Be able to implement thread classes, create thread objects, and initiate and terminate threads
- Understand the concepts of concurrency and synchronization and how to create thread safe classes
- Be able to explain the Producer-Consumer synchronization problem connected with threads and its solution
- Know how to use thread safe Java API classes within a multithreaded application

14.1 OVERVIEW

In computer science, *concurrency* is the concept of executing several programs, or several parts of a program, at the same time or giving the user the impression that they are being executed at the same time. For example, one program may be displaying and updating the time of day while another program is playing a music video available on a Website. In addition, one part of the music program could be downloading the video from the Website while another part of that program is playing the part of the video already downloaded.

Concurrently executing programs, such as a time-of-day program and a program that plays music videos are often referred to as *processes*. Concurrently executing parts of a process, such as the two parts of a music video program that downloads and simultaneously plays the video, are referred to as *threads*. Figure 14.1 shows two processes in concurrent execution. Even what appears to be a single program running on a computer system could be a process with a set of simultaneously executing threads. In fact, when we consider the support every program receives from the operating system and the Java Runtime Environment, every Java program is a process with multiple threads.

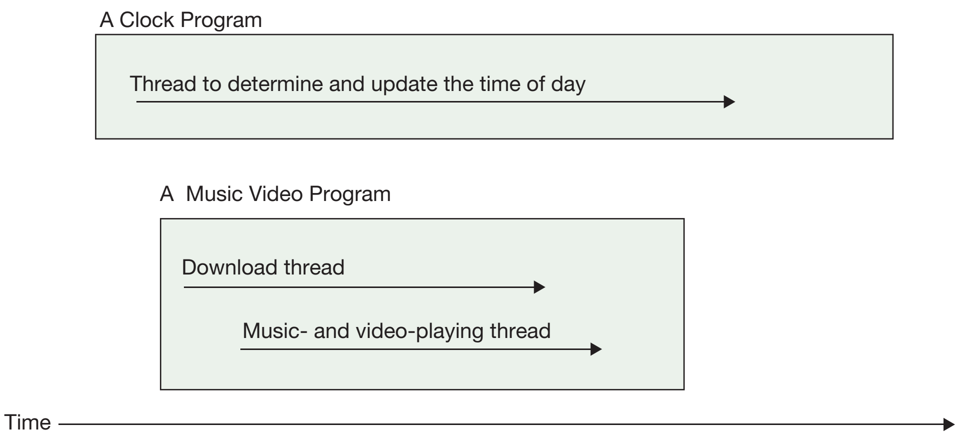


Figure 14.1
A single-thread and a double-thread process.

A thread can be considered to be an execution path through a program, and the programs discussed up to this point in the text should be thought of as having one thread. This thread, called the *main thread*, contains the code of the method `main`, and it is created and started by the Java Runtime Environment when the program is launched. As we will see, the main thread can create and start other threads. The creation and starting of threads, and the sharing of information and system resources among them, are the topics of concurrent (*multitasking*) programming.

Actual and Perceived Concurrency

For a thread to be in execution, the address of one of its instructions must be stored in the instruction register of a CPU in order for the instruction to be interpreted (decoded) and executed by the CPU's data path. As a result, two or more threads can only execute concurrently if the system they are running on has more than one CPU or the system's CPU contains multiple data paths, called cores. A system that has one CPU containing one data path can give the impression that several threads are executing simultaneously by sharing the CPU's processing time among threads.

Threads that share a CPU's data path are each given a quantum (or "time slice") of CPU time based on a scheduling algorithm, which is platform or operating system dependent. Fortunately, the speeds of modern computers permit a significant number of instructions, often in the millions, to execute during a time quantum. In addition, the time interval between the quanta is small enough to give the system user the perception that all of the threads are being executed without interruption. This is analogous to our perception that an incandescent light bulb does not blink sixty times a second.

Several of the popular sorting algorithms, such as the Merge Sort and the Quick Sort, can be coded concurrently. They both divide the lists of items to be sorted into sublists which can be sorted simultaneously.

14.2 CREATING AND INITIATING THREADS

There are three ways to create and initiate the execution of a thread. All of them involve coding a class that includes a method named `run` defined in the interface `Runnable`. When a thread is allocated its first quantum of CPU time, its execution begins with the first executable statement in the method `run`. This method is to a thread as the method `main` is to an application. The thread's algorithm is coded in this method.

Defining a Thread's Class

A thread class's heading must either indicate that it implements the interface `Runnable` or extends the class `Thread` (which implements the interface `Runnable`). Figure 14.2 shows a thread class named `ExtendsThread` that extends the class `Thread`, and Figure 14.3 shows a class named `ImplementsRunnable` that implements the interface `Runnable`. As shown in Figure 14.2, the first executable statement in the constructor of a class that extends `Thread` should be an invocation of its parent's constructor. Both classes implement the method `run`, defined in the interface `Runnable`, and contain the `String` data member `name` that is initialized to the string passed to the class's constructor.

```
import javax.swing.*;

public class ExtendsThread extends Thread
{
  private String name;

  public ExtendsThread(String name)
  {
    super();
    this.name = name;
  }

  public void run() //A thread's entry point
  {
    System.out.println(name + " is executing");

    String answer  = JOptionPane.showInputDialog("What is 23 + 57 ?");
    if(answer.equals("80"))
    {
      System.out.println("Correct, 23 + 57  = 80");
    }
    else
    {
      System.out.println("Incorrect, 23 + 57  = 80");
    }
  }
}
```

Figure 14.2
The class **ExtendsThread** that extends the class **Thread**.

```
public class ImplementsRunnable implements Runnable
{
  private String name;
  private int nLines;

  public ImplementsRunnable(String name)
  {
    this.name = name;
  }

  public void run() //A thread's entry point
  {
    System.out.println(name + “ is executing”);
  }
}
```

Figure 14.3
The class **ImplementsRunnable** that implements the interface **Runnable**.

Initiating a Thread's Execution

The code of the `main` method creates an object in the thread's class. The next step is to initiate this object's execution from within the `main` method, which makes the thread eligible for a quantum of CPU time. The way this is done depends on whether the thread class's heading indicates that it extends the class `Thread` or implements the interface `Runnable`.

If the class extends the class `Thread` and is named `ExtendsThread` (as in Figure 14.2), the following code fragment would be used in the `main` method to create the `Thread` instance `thread1` and make the thread eligible for a quantum of CPU time:

```
//Create and initiate a thread whose class extends Thread
ExtendsThread thread1 = new ExtendsThread("thread1");
thread1.start();
```

We will learn more about the `Thread` class's `start` method in the next section. When any thread is granted its first quantum of CPU computing time, its execution begins with the first executable statement in its `run` method.

If the thread's class implements the class `Runnable` (as in Figure 14.3), there are two alternative approaches to initiate the thread's execution. Assuming the thread class is named `ImplementsRunnable`, the following code fragment would be used in the `main` method to create the `Thread` instance `thread2` and make the thread eligible for a quantum of CPU time. It creates a `Runnable` object and passes it to the one-parameter constructor of the `Thread` class.

```
// Create and initiate a thread whose class implements Runnable
ImplementsRunnable runnableObj = new ImplementsRunnable("thread2");
Thread thread2 = new Thread(runnableObj);
thread2.start();
```

The alternate technique when the thread class implements the interface `Runnable` is to not explicitly declare the `Thread` instance or initiate its execution. This approach uses the concept of an *executor service* to create the thread and manage the initiation and execution of the thread. Using this approach, we can reduce the amount of overhead associated with creating multiple threads because an executor maintains an expandable pool of threads, each of which can be assigned or *reassigned* to a `Runnable` object.

The following code fragment illustrates this approach. It invokes the `Executors` class's static method `newCachedThreadPool` to create an executor service referenced by `threadLauncher`. This object maintains a thread pool and is capable of adding threads to the pool and assigning runnable objects to existing threads. Its `execute` method assigns the `Runnable` object passed to its parameter (in our case, an instance of the class `ImplementsRunnable`) to one of the threads in its pool and initiates its execution.

```
ImplementsRunnable runnableObj = new ImplementsRunnable("thread3");
ExecutorService threadLauncher = Executors.newCachedThreadPool();
threadLauncher.execute(runnableObj); //initiate thread3 as runnableObj
```

The application `CreatingThreads`, shown in Figure 14.4, creates and initiates three threads using the three techniques discussed in this section. The output produced by the threads is shown in Figure 14.5.

The first thread, created on line 8, is an instance of the class `ExtendsThread` shown in Figure 14.2, which as its name implies, extends the class `Thread`. The second thread is created on line 12 by passing the object created on line 11, `runnableObj1`, to `Thread's` one-parameter constructor. The object `runnableObj1` is an instance of the class `ImplementsRunnable` shown in Figure 14.3, which implements the class `Runnable`. These two threads are initiated on lines 19 and 20 of Figure 14.4.

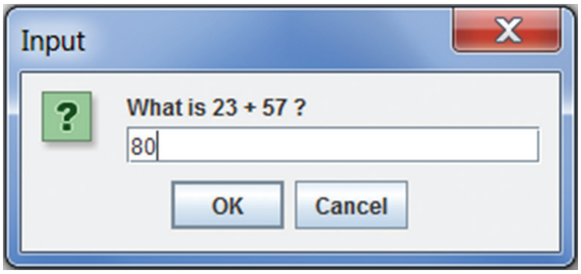
To initiate the third thread, another instance of the class `ImplementsRunnable` (Figure 14.3) is created on line 15, the instance `runnableObj2`. Line 23 passes this object to the `execute` method invoked on `threadLauncher`: the object created on line 16 that contains a thread pool. The invocation of `execute` on line 23 causes a thread to be added to this thread pool, the object `runnableObj2` is associated with the thread, and the thread is initiated.

These three threads and the main thread will share CPU time during the program's execution. As soon as a thread receives a quantum of computing time, the first statement in its run method (lines 15 and 13 of Figures 14.2 and 14.3, respectively) output the thread's name data member to the system console. The output sequence shown in the lower portion of Figure 14.5 indicates that the main method ended before any of the threads were granted a quantum of computing time. The output also indicates that `thread1` was granted a quantum of computing time before `thread2`, which was granted a quantum of computing time before the third thread. The last console output shown at the bottom of the figure was generated after the user entered 80 in the input dialog box displayed by `thread1` (shown in the upper portion of the figure) and then clicked OK. At this point, the other two threads had already completed their execution.

```
1    import java.util.concurrent.*;
2
3    public class CreatingThreads
4    {
5      public static void main(String[] args)
6      {
7        // create a thread
8        ExtendsThread thread1 = new ExtendsThread(“thread1”);
9
10       // create a runnable object and then a thread
11       ImplementsRunnable runnableObj1 = new ImplementsRunnable(“thread2”);
12       Thread thread2 = new Thread(runnableObj1); //creates a thread
13
14       // create a runnable object and a thread pool
15       ImplementsRunnable runnableObj2 = new ImplementsRunnable(“thread3”);
16       ExecutorService threadLauncher = Executors.newCachedThreadPool();
17
18       // initiate the threads
19       thread1.start(); //initiates thread 1
20       thread2.start(); //initiates thread 2
21
22       // assign a runnable object to a thread in the thread pool
```

```
23        threadLauncher.execute(runnableObj2);
24
25        threadLauncher.shutdown();
26
27        System.out.println("main method has completed its execution");
28     }
29  }
```

Figure 14.4
The application **CreatingThreads**.



Console Output:
main method has completed its execution
thread1 is executing
thread2 is executing
thread3 is executing
Correct 23 + 57 = 80

Figure 14.5
An input to the application **CreatingThreads** and the outputs generated by it.

14.3 THREAD STATES

The time from when a thread object is created until it no longer exists is called a thread's *lifetime*. During its lifetime, a thread is in one of six Java-defined states within the Java Virtual Machine. The names of the six states are: new, runnable, blocked, waiting, timed waiting, and terminated. Figure 14.6 shows these states and the possible transitions from one state to another depicted by the arrows in the figure. The green arrows indicate that the thread is becoming more likely to being granted a quantum of computing time.

14.3.1 The New, Runnable, and Terminated States

As shown on the left side of Figure 14.6, when a thread is created, it enters the new state. It remains there until it is initiated by the method `start` or the method `execute` (e.g., lines 19 and 23 of Figure 14.4, respectively). Once initiated, it enters the runnable state. Only threads that are in the runnable state can be assigned a quantum of CPU computing time by the operating system. When the thread completes its execution, it enters the terminate state. All system resources allocated to a task that is in this state are reclaimed, and the task ceases to exist (shown in the lower right side of Figure 14.6).

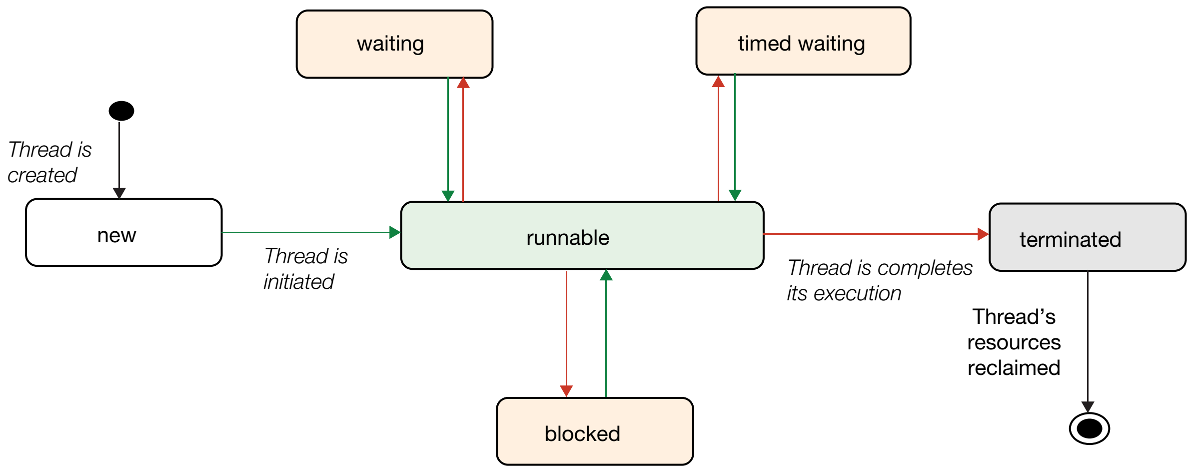


Figure 14.6
Transitions among the six states of a thread during its life.

Threads that are in the runnable state can be assigned a quantum of computing time by the operating system. The algorithm used to select which of the threads in the runnable state receives the next quantum of CPU time is platform dependent and is normally based on a priority assigned to the thread. The higher the task's priority, the more likely it is to receive a quantum of computing time when it is in the runnable state. Round robin scheduling is a common algorithm used by operating systems to assign a quantum of computing time to tasks of equal priority while they are in the runnable state. Figure 14.7 shows this process for three tasks of equal priority named thread1, thread2, and thread3.

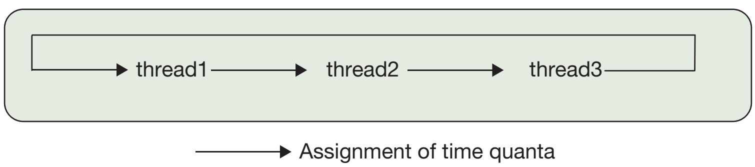


Figure 14.7
Round robin scheduling of three threads in the runnable state.

By default, Java assigns the newly created thread the priority of the thread that created it. When an application's `main` method is created, its priority is set to the `Thread` class's static integer constant, `NORM _ PRIORITY`, which is midway between the class's constants `MAX _ PRIORIY` and `MIN _ PRIORITY`. This default priority value can be changed by invoking the `Thread` class's `set-Priority` method and passing it a new priority within the range of the static constants `Thread.MAX _ PRIORITY` to `Thread.MIN _ PRIORITY`, inclusive.

14.3.2 The Blocked, Waiting, and Timed Waiting States

Most threads over their lifetime transfer between the runnable state into the waiting, timed waiting, or blocked states several times. These transitions can be initiated by various events, such as the thread performing I/O or by threads invoking methods that initiate the transfer.

The Blocked State

After a thread has issued a request for input (e.g., displayed an input prompt to the user and is waiting for the completion of the input), it is moved from the runnable state into the blocked state because it cannot continue its execution until the input is complete. This transfer from the runnable state into the blocked state and back again, shown in the bottom center of Figure 14.8, is associated with the initiation and completion of an input event.

A thread can also enter the blocked state if it contains a Java concurrency construct called a *synchronized code block*, but it cannot enter the code block if a warning, called a *lock,* has been issued indicating it is unsafe to execute the code. For example, if the task of a thread's synchronized code block was to increase the pressure inside a storage tank, another thread whose task was to determine if the tank's pressure sensor was functional could issue a lock until it determined that the sensor was functional. Synchronized code blocks are also used to prevent one thread from changing shared data while another thread is reading it.

The transition from the runnable state to the blocked state and back again, initiated by a locked and then unlocked synchronized code block, is also depicted in the bottom center portion of Figure 14.8. We will learn more about synchronized code blocks and locks later in this chapter.

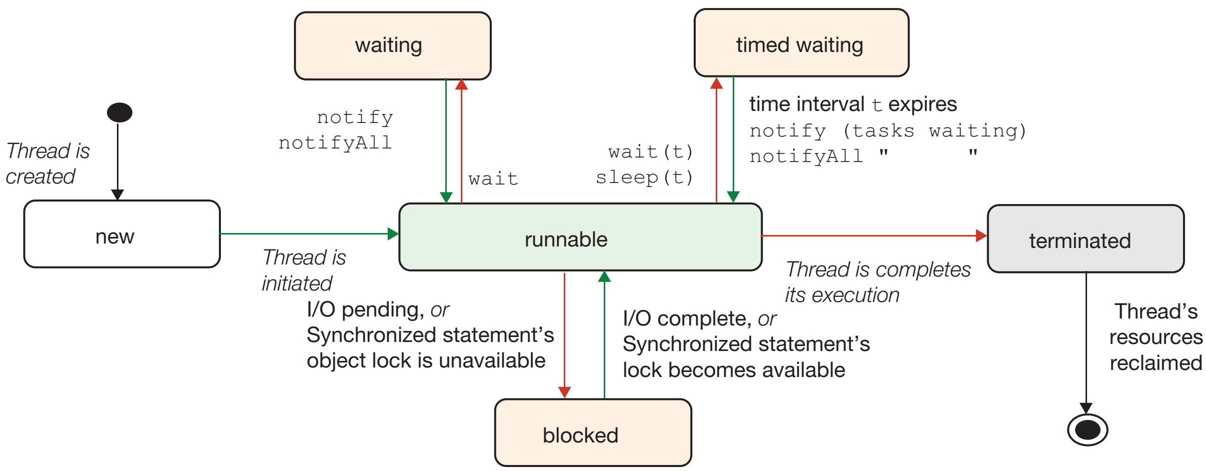


Figure 14.8
Actions and events that transition a thread from the runnable state and back again.

The Waiting and Timed Waiting States

A thread moves from the runnable state to the waiting or timed waiting state when it, or another thread, invokes the `wait` method on the thread. It can also enter the timed waiting state when the `sleep` method is invoked on it. These state transitions are shown in the top center portion of Figure 14.8.

When a thread in the runnable state invokes the `wait` method and does not pass it an argument, the thread enters the waiting state. A thread that does not have all the system resources it needs to complete its algorithm, such as when a piece of data or a resource it needs is not yet available, will enter the waiting state via an invocation of the no-parameter version of the method `wait.` The thread will reenter the runnable state when another thread *notifies* it or notifies all of the threads waiting for the data item or resource, that the data item or resource is available via an invocation of the `notify` or `notifyAll` methods. This transition back to the runnable state is depicted at the top-left portion of Figure 14.8.

When a thread in the runnable state invokes the `wait` method or the `Thread` class's static method `sleep` and passes it an integer argument, the thread enters the timed waiting state for that number of milliseconds. When the time period expires, the thread returns to the runnable state, as shown in the top right portion of Figure 14.8. A thread that cannot perform its task until a known period of time passes can use either the `sleep` or the `wait` method to exit the runnable state. The following code fragment would be used by a task to remove itself from the runnable state for one second (i.e., 1000 milliseconds):

```
Thread.sleep(1000);
```

Threads placed in the timed waiting state with an invocation to `wait` also exit the state when an invocation to the `notify` or `notifyAll` method is made by some other thread. If a thread was willing to wait for up to one minute for a piece of data that was not yet available, and then proceed with or without the data item, it would do so using the following code fragment:

```
int t = 1000 * 60; //one minute = 60,000 milliseconds
wait(t); //in the timed waiting state
```

Concurrent algorithms that use the methods `wait`, `sleep`, `notify`, and `notifyAll` to perform state transitions are quite common, however, an error-free implementation of these algorithms can be rather elusive. As a result, Java provides several levels of support to facilitate their use. Armed with an understanding of how to create threads, thread states, and the events and methods that initiate transitions between states, we are now ready to expand our concurrency knowledge base by implementing a classic concurrent algorithm: the Producer-Consumer algorithm, which if not properly implemented results in the Producer-Consumer problem.

14.4 THE PRODUCER-CONSUMER PROBLEM

The Producer-Consumer problem is a classic concurrency synchronization problem that arises when one thread is producing (i.e., generating) a resource that another task is consuming (i.e., using). For example, one thread is computing a data value that another thread outputs to the system console after combining it with other data values. The word consumer is used in the name of the problem because an assumption of the problem is that once the resource (e.g., a data item) is obtained by the thread that uses it, the thread that produced it no longer needs to maintain a record of it. (This situation was made famous by the Pac-Man game.)

In object oriented programming, data is shared between threads using an object whose address is known to both the producer and consumer threads. The shared resource is typically a data member of the object whose class implements a `set` method the producer method invokes and a `get` method the consumer thread invokes. The object is referred to as a *buffer object*, or more simply a *buffer*, because it is a holding area for the shared resource. A *bounded buffer* is a buffer object that can hold a specified number of shared items. An object in the API class `ArrayBlockingQueue`, which we will learn more about later in this chapter, can be used as a bounded buffer to safely share data between producer and consumer tasks.

The class `Buffer`, shown in Figure 14.9, is an example that buffers an integer data item in its data member `sharedData`, defined on line 3. The single thread application `BufferDemo`, shown in Figure 14.10, uses the `Buffer` object `aBuffer` to store three integer data items it alternately produces (stores in the buffer's data member) on line 10 and then consumes (fetches) and outputs on line 13. The output generated by the program is shown in Figure 14.11.

```
1   public class Buffer
2   {
3     private int sharedData;
4
5     public Buffer()
6     {
7
8     }
9
10    public void setData(int dataItem)
11    {
12      sharedData = dataItem;
13    }
14
15    public int getData()
16    {
17      return sharedData;
18    }
19  }
```

Figure 14.9
The class **Buffer**.

```
1   public class BufferDemo
2   {
3     public static void main(String[] args)
4     {
5       private Buffer aBuffer = new Buffer();
6       private int dataItem;
7
8       for(int i = 1; i <= 3; i++)
9       {
```

```
10        aBuffer.setData(i); //produce a data item
11        System.out.println("Produced " + i);
12
13        dataItem = aBuffer.getData(); //consume a data item
14        System.out.println("Consumed " + dataItem);
15      }
16    }
17  }
```

Figure 14.10
The application **`BufferDemo`**.

```
Produced 1
Consumed 1
Produced 2
Consumed 2
Produced 3
Consumed 3
```

Figure 14.11
The output produced by the application **`BufferDemo`**.

Although the application `Buffer` does produce and consume integer data items stored in a `Buffer` object, it does not illustrate the producer and consumer problem because the buffer is not shared between two threads. The main thread is the only thread in the application. It does not simulate a producer thread performing a second write to the buffer before a consumer thread is issued a quantum of computing time to read the first item written to the buffer. In this single thread example, once line 10 of Figure 14.10 executes, it cannot execute again until line 13 executes.

There is one case when this single thread producer-consumer application could degenerate into a producer-consumer problem: if lines 10 and 11 of Figure 14.10 were switched with lines 13 and 14. In this case, the consumer would be consuming its first data item before the producer produced it, and the last data item produced (3) would not be consumed. Figure 14.12 shows the application's output when lines 10–11 and lines 13–14 are switched. As shown in the figure, the first item consumed is the default value of the buffer object's integer data member `sharedData`, 0, which is not a produced value, and the last data item consumed is 2. The data item 3 is not consumed.

```
Consumed 0
Produced 1
Consumed 1
Produced 2
Consumed 2
Produced 3
```

Figure 14.12
The output produced by the application **`BufferDemo`** when lines 10–11 are switched with lines 13–14.

While this error in the application would be easily discovered and rectified because the erroneous output is repeatable and the producer and consumer are in the same thread (`main`'s thread), this type of problem is much more difficult to avoid, discover, and rectify when the producer and consumer are in different threads or in entirely different applications. The complications include the fact that the output becomes less repeatable or predictable in a multithreaded version of the application, and the producer and/or consumer's time quantum could expire before their invocations of the `set` and `get` methods complete execution.

To illustrate these complications, the producer and consumer tasks, lines 10–11 and 13–15 of Figure 14.10, have been transferred into two different classes named `Producer` and `Consumer`, shown in Figures 14.13 and 14.14, respectively. These classes implement the interface `Runnable`, so instances of these classes can be used to separate the producer and consumer tasks into two separate threads.

When a `Producer` object is created, the class's constructor (line 8 of Figure 14.13) is passed an instance of the class `Buffer` whose code is shown in Figure 14.9. The address of the `Buffer` class object is stored in the data member `sharedData` declared on line 5. The producer tasks coded as lines 24 and 25 are inside a `for` loop (lines 15–26) that executes ten times. The loop variable, `i`, is passed to the shared `Buffer` object's `setData` method on line 24. A producer object will produce the values 1 to 10 in ascending order.

Line 19 invokes the `Thread` class's static method `sleep` to move the `Producer` object into the timed waiting state for a random time period of up to ten milliseconds. This is to simulate a random number of time quanta expiring during the processing required to produce the shared data item, `i`. The `Random` object `delay` used to generate the time increment passed to the `sleep` method is declared on line 6. The `sleep` method can throw a checked `InterruptedException` object, which is why it is coded inside a `try-catch` construct (lines 17–23).

The code block of the class `Consumer`, shown in Figure 14.14, is similar to that of the `Producer` class's code block. The most obvious difference is that producer tasks (lines 24–25 of Figure 14.13) have been replaced by the consumer tasks (lines 28 and 29 of Figure 14.14). In addition to this change, before the consumers `run` method ends, it invokes the method `outputConsumedSummary` (line 35). This method outputs two lines of statistics that summarize the performance of a `Consumer` thread. The two arrays it uses to store the statistics are declared on lines 6 and 7 and updated each time through the run method's `for` loop on lines 32 and 33.

```
1   import java.util.Random;
2
3   public class Producer implements Runnable
4   {
5     private Buffer sharedData;
6     private Random delay = new Random();
7
8     public Producer(Buffer sharedData)
9     {
```

```
      this.sharedData = sharedData;
    }

    public void run()
    {
      for(int i = 1; i <= 10; i++)
      {
        try
        {
          Thread.sleep(delay.nextInt(10) + 1); //simulate data processing
        }
        catch(InterruptedException e)
        {
        }
        sharedData.setData(i);
        System.out.println("Produced " + i);
      }
    }
  }
```

Figure 14.13
The class **Producer**.

```
import java.util.Random;

public class Consumer implements Runnable
{
  private Buffer sharedData;
  private int[] timesConsumed = new int[10];
  private boolean[] consumedData = new boolean[10];

  public Consumer(Buffer sharedData)
  {
   this.sharedData = sharedData;
  }

  public void run()
  {
    Random delay = new Random();
    int dataItem;

    for(int i = 1; i <= 10; i++)
    {
      try
      {
        Thread.sleep(delay.nextInt(10) + 1); //simulate data fetch
      }
```

```
      catch(InterruptedException e)
      {
      }
      dataItem = sharedData.getData();
      System.out.println("Consumed " + dataItem + " <---");

      //record consumed statistics
      consumedData[dataItem - 1] = true;
      timesConsumed[dataItem - 1]++;
    }
    outputConsumedSummary();
  }

  private void outputConsumedSummary() //outputs final statistics
  {
    try
    {
      Thread.sleep(5000);
    }
    catch(InterruptedException e)
    {
    }
    System.out.print("Consumed data: ");
    for(int i = 1; i <= 10; i++)
    {
      if(consumedData[i-1] == true)
      {
        System.out.print(" " + i);
      }
    }
    System.out.print("\nTimes consumed:");
    for(int i = 1; i <= 10; i++)
    {
      if(consumedData[i-1] == true)
      {
        System.out.print(" " + timesConsumed[i-1]);
      }
    }
  }
}
```

Figure 14.14
The class **Consumer**.

The application `PCThreadProblems` presented in Figure 14.15 declares an instance of a `Producer` and a `Consumer` object on lines 10 and 11, passing their constructors the shared `Buffer` object declared on line 8. Then, lines 13–15 associate these runnable objects with threads and initiate them. The output produced by two successive executions of the application is shown on the left and right sides of Figure 14.16. Line numbers were added to the figure to facilitate its discussion.

If there were no producer-consumer problems, both columns of output would be the same as the output shown in Figure 14.17, in which all of the values produced are immediately followed by the value being consumed. Each value produced (the values 1 through 10) is consumed once and only once, as indicated by the summary at the bottom of the figure. As the summary at the bottom of each column of Figure 14.16 indicates, not all of the data produced by the two executions of the application `PCThreadProblems` were consumed. On the left side of the figure, only the values 1, 3, 5, 6, and 7 were consumed, with 1 being consumed twice and 3 and 6 being consumed three times each. Similar problems occurred during the second execution of the application, as shown on the right side of the figure.

These problems occur in this producer-consumer multithreaded application because the producer and consumer are not waiting for each other to complete their tasks. Whenever either thread receives a quantum of computing time, it produces or consumes as many values as it can, without any consideration of whether or not the other thread has consumed or produced a value. Referring to lines 13–15 of the left column of Figure 14.16, the consumer thread was able to consume the same value (6) from the buffer three times before the producer thread could produce the value 7 (line 16). In some cases, the producer thread was able to produce several values before the consumer thread was able to consume them, as shown on lines 7–9 of the right column in Figure 14.16.

The obvious remedy is for the producer to produce a value and not produce another value until the consumer consumes the value. Similarly, once the consumer consumes a value, it should not consume another value until the producer produces another value. This process is referred to as synchronizing the producer and consumer tasks. Synchronizing these tasks assures that the producer and consumer will alternate their access to the buffer, as depicted in Figure 14.17.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PCThreadProblems
{
  public static void main(String[] args)
  {
    Buffer aBuffer = new Buffer();

    Producer producerThread = new Producer(aBuffer);
    Consumer consumerThread = new Consumer(aBuffer);

    ExecutorService launcher = Executors.newCachedThreadPool();
    launcher.execute(producerThread);
    launcher.execute(consumerThread);

    launcher.shutdown();
  }
}
```

Figure 14.15
The application **`PCThreadProblems`**.

```
Produced 1                        Produced 1
Consumed 1 <---                   Consumed 1 <---
Consumed 1 <---                   Produced 2
Produced 2                        Consumed 2 <---
Produced 3                        Produced 3
Consumed 3 <---                   Consumed 3 <---
Consumed 3 <---                   Produced 4
Consumed 3 <---                   Produced 5
Produced 4                        Produced 6
Produced 5                        Consumed 6 <---
Consumed 5 <---                   Produced 7
Produced 6                        Produced 8
Consumed 6 <---                   Consumed 8 <---
Consumed 6 <---                   Produced 9
Consumed 6 <---                   Consumed 9 <---
Produced 7                        Consumed 9 <---
Consumed 7 <---                   Produced 10
Produced 8                        Consumed 10 <---
Produced 9                        Consumed 10 <---
Produced 10                       Consumed 10 <---
Consumed data:  1 3 5 6 7         Consumed data:  1 2 3 6 8 9 10
Times consumed: 2 3 1 3 1         Times consumed: 1 1 1 1 1 2 3
```

First Execution **Second Execution**

Figure 14.16
The output produced by two successive executions of the application **`PCThreadProblems`**.

```
Produced 1
Consumed 1 <---
Produced 2
Consumed 2 <---
Produced 3
Consumed 3 <---
Produced 4
Consumed 4 <---
Produced 5
Consumed 5 <---
Produced 6
Consumed 6 <---
Produced 7
Consumed 7 <---
Produced 8
Consumed 8 <---
Produced 9
Consumed 9 <---
Produced 10
Consumed 10 <---
Consumed data:  1 2 3 4 5 6 7 8 9 10
Times consumed: 1 1 1 1 1 1 1 1 1 1
```

Figure 14.17
A problem-free producer-consumer output.

14.5 SOLUTIONS TO THE PRODUCER CONSUMER PROBLEM

One approach to solving producer-consumer problems revealed by the application shown in Figure 14.15 is to replace the `Buffer` class instance declared on line 8 of Figure 14.15 with an instance of a synchronized API collection class. These classes are said to be thread safe. API collection classes that are thread safe do not allow a consumer to consume unless a producer has produced and vice versa.

We will discuss this approach after we discuss and implement changes to the `Buffer` class that would make it thread safe. In this thread safe buffer implementation, the `wait` and `notify` methods and a Java lock are used to synchronize a producer's and consumer's access to the shared data object. The changes we will make to the `Buffer` class are effectively what we would see if we looked "under the hood" of the API's implementations of its thread safe collection classes.

14.5.1 Synchronizing a Buffer Class: Synchronized Methods

The changes we will make to the `Buffer` class to make it thread safe involve modifications to the class's `setData` and `getData` methods that are based on the following three synchronization criteria:

- Its `getData` method should have a way of determining if a *new* data item has been written to the buffer and only return a value when this is the case
- The `setData` method should have a way of determining if the data item currently in the buffer has been *consumed* and only overwrite the data item when this is true
- When one method is in execution, the other method should not be allowed to begin execution; that is, they should not be permitted to execute concurrently, they should be mutually exclusive

The first two criteria will involve adding two Boolean data members to the `Buffer` class, which we will name `writeable` and `readable`. These variables will be used by the `setData` and `getData` methods to determine if they should write to, or read from, the buffer. The `setData` method will write to the buffer when `writeable` is `true`, and the `getData` method will read from the buffer when `readable` is `true`.

Before the methods end their execution, they will reverse the truth values of the two variables:

- `setData` will set `readable` to `true` to indicate to `getData` that it can read from the buffer, and it will set `writeable` to `false` to remind itself that it cannot write to the buffer
- `getData` will set `writeable` to `true` to indicate to `setData` that it can write to the buffer, and it will set `readable` to `false` to remind itself that it cannot read from the buffer

The code fragments shown in Figure 14.17 illustrate these ideas but should be considered pseudocode. They will be modified when incorporated into the revised `Buffer` class because they do not satisfy our third criteria: when one method is in execution, the other method should not be allowed to begin execution.

setData method

```
1 if(writeable == true)
2 {
3   //write the buffer's data member
4    writeable = false;
5    readable = true;
6 }
```

getData method

```
1 if(readable == true)
2 {
3    writeable = true;
4    readable = false;
5    //return buffer's data member
6 }
```

Figure 14.18
Pseudocode of the first two producer-consumer synchronization criteria.

The need for the third criterion is more easily understood in the context of the first two criteria, which are expressed in Figure 14.18. Consider the case when the code fragment of the `getData` method shown on the right side of the figure has just completed line 3, setting `writeable` to `true`. If at this point the `setData` method was allowed to execute, the Boolean condition on line 1 (on the left side of figure) would evaluate to `true`, and line 3 would write a new value into the buffer. Here's the problem: line 5 of the `getData` method has not yet executed to return the value that was just overwritten. Allowing this situation to occur results in a lost data item, i.e., a produced data item that is not consumed. Other execution-sequence scenarios result in the other equally unacceptable outcome: a data item is consumed twice. It is often critical that the execution of a portion of a thread's code not be interrupted once it begins. This section of code is referred to as a *critical section*.

Java provides a remedy for this situation by allowing us to declare methods to be synchronized. When two or more methods in a class are declared synchronized and one of the methods is in execution, the other method(s) cannot begin execution. A thread that attempts to initiate the execution of a second synchronized method enters the waiting state. Declaring the `setData` and `getData` methods to be synchronized would solve the lost data item problem we cited previously. When the methods are synchronized after the `getData` method sets `writeable` to `true` on line 3 (right side Figure 14.18), the `setData` method cannot begin its execution until line 5 of the `getData` method executes and returns the now non-overwritten data value to its invoker. The producer thread that invoked the `setData` method would enter the waiting state.

There are two subtle, but remaining pieces to the puzzle that the following two questions expose:

- How does the producer thread that was moved to the waiting state when it invoked `setData` while `getData` was in execution return to the runnable state?
- What if the `getData` method is invoked while `readable` is `false`, in which case its `return` statement (line 5 of Figure 14.18) is unreachable? (The Java version of the pseudocode would produce a translation error.)

The solution to the first remaining piece of the puzzle is that the `getData` method invokes either the `notify` method (or `notifyAll` method) just before it ends. This moves one (or all) of the threads that invoked methods synchronized to the `getData` method from the waiting state to the runnable state. In our case, the `setData` method would then begin its execution when the operating system granted the thread that invoked it a CPU time quantum. A similar modification, an invocation of the method `notify` or `notifyAll`, must be added to the `setData` method.

The solution to the second remaining piece of the puzzle (what if the `getData` method is invoked while `readable` is `false`, in which case its return statement is unreachable) also has to do with a thread entering the waiting state. In this case, the thread that invoked the `setData` method effectively places itself into the waiting state to pause its execution until `readable` becomes `true`. It does this by invoking the `wait` method inside the `getData` method. The thread will return to the runnable state when a thread that invoked the `setData` method issues an invocation to `notify` or `notifyAll`. A similar modification has to be made to the thread that invokes the `setData` method.

Synchronizing the `setData` and `getData` methods also prevents another facet of the producer-consumer problem from occurring: *deadlock*. Deadlock occurs when two or more threads, which share resources, are all waiting for each other to complete and none of them can proceed. The situation is analogous to north bound and east bound cars at the stop signs of an intersection with their drivers waiting for each other to proceed, or north and south bound cars are waiting for each other at a one lane bridge. To see when this would occur in non-synchronized methods, let us once again consider the case when the code fragment of the `getData` method shown on the right side of Figure 14.18 has just completed line 3, setting `writeable` to `true`. If at this point, the `setData` method was allowed to execute and run to completion, line 4 of the method would set `writeable` back to `false`. After the method ended, the `getData` method would continue its execution at line 4, and `readable` would be set to `false`. Now both `writeable` and `readable` are `false` and neither method can execute.

NOTE *When writeable data is being shared between two or more threads, the access to the data must be synchronized.*

Synchronized Methods

Methods are declared synchronized by including the keyword `synchronized` in their signature. The execution of a class's synchronized methods is mutually exclusive, and a lock analogy is used to explain the transfer of execution from one synchronized method to another.

One lock is shared by all of the synchronized methods, and initially it is available to all of the synchronized methods. When a synchronized method is invoked by a thread, it must first acquire the lock for it to begin execution. Once acquired, the lock effectively locks out all invocations of other synchronized methods issued by threads. When an invocation is issued by a thread and the lock is not available (i.e., another synchronized method has previously acquired the lock), the thread enters the waiting state until the method in execution surrenders the lock. A method implicitly surrenders the lock when it invokes the `wait` method, the `notify` method, or the `notifyAll` method.

The Synchronized Buffer Class SynchronizedBuffer

The class `SynchronizedBuffer`, shown in Figure 14.19, is the thread safe version of the class `Buffer` presented in Figure 14.9. It uses synchronized methods and other concepts discussed in this section to eliminate the producer-consumer problems resulting from the use of the `Buffer` class by the application `PCThreadProblems` (Figure 14.15).

The `SynchronizedBuffer`'s revised `setdata` and `getData` methods begin on lines 12 and 31 of Figure 14.19, respectively. Their signatures include the keyword `synchronized`, which declares them to be synchronized methods and making their executions mutually exclusive. Lines 25–27 of the `setData` method and lines 45–47 and line 49 of the `getData` method are analogous to lines 3–5 on the left and right sides of Figure 14.18, respectively. This is the portion of the methods that access the shared buffer and reverse the truth values of the Boolean variables `writeable` and `readable` declared on lines 4 and 5 of Figure 4.19.

The `while` statements at the beginning of these methods (lines 16 and 36), use the variables `writeable` and `readable` in their Boolean conditions. When these variables are `false`, the methods place themselves in the waiting state by invoking the `wait` method (lines 18 and 38). This prevents the `return` statement on line 49 from being unreachable, and also surrenders the lock.

The variable `readable` is initialized to `false` on line 5. This guarantees that if a consumer thread executes before a producer thread and the `getData` method acquires the lock, it will execute the `wait` statement on line 38, surrender the lock, and enter the waiting state. Then, when a producer thread executes and invokes the `setData` method, the method can acquire the lock. Because `writeable` is initialized to `true` on line 4, the producer method's execution proceeds to completion (lines 25–28) producing a data item, reversing the truth values of the Boolean variables, and finally invokes `notifyAll` to surrender the lock.

After `notifyAll` is invoked, the consumer task will reenter the runnable state, and its pending invocation of `getData` can reacquire the lock. Because the `setData` method set `readable` to `true`, the `getData` proceeds to completion. Before it ends, it reverses the truth value values of the Boolean conditions, invokes `notifyAll`, and returns the consumed data item its invoker.

Instead of the `getData` method returning the buffer's data item `sharedData` on line 49, it returns the contents of the local variable `dataItem`. This local variable was assigned `sharedData`'s value on line 47. This is done because the invocation to `notifyAll` has to be coded before the `return` statement on line 49, or it will be unreachable. Here's the problem: once `notifyAll` is invoked, the producer thread could execute and invoke the `setData` method. Because `writeable` has already been set to `true` (line 45) the `setData` method will execute to completion and overwrite the contents of the variable `sharedData` before line 49 executes. If this happened and line 49 returned the variable `sharedData`, the *new* data item written would be returned, and a data item would be lost. By returning the variable `dataItem`, the `getData` method correctly returns the potentially overwritten data item.

```
1   public class SynchronizedBuffer
2   {
3     int sharedData;
4     private boolean writeable = true;
5     private boolean readable = false;
6
```

```
  public SynchronizedBuffer()
  {

  }

  public synchronized void setData(int dataItem)
  {
    try
    {
      while(writeable == false)
      {
        wait(); // releases the lock on synchronized methods
      }
    }
    catch(InterruptedException e)
    {
    }

    sharedData = dataItem;
    writeable = false;
    readable = true;
    notifyAll();
  }

  public synchronized int getData()
  {
    int dataItem;
    try
    {
      while(readable == false)
      {
        wait();
      }
    }
    catch(InterruptedException e)
    {
    }

    writeable = true;
    readable = false;
    dataItem = sharedData;
    notifyAll();
    return dataItem;
  }
}
```

Figure 14.19
The class **SynchronizedBuffer**.

In the interest of completeness, the changes to the original `Producer` and `Consumer` classes (presented in Figures 14.13 and 14.14) necessary for them to share data via an instance of the class

`SynchronizedBuffer` are given as highlighted lines of code in Figures 14.20 and 14.21. Aside from their class names being changed to `ProducerV2` and `ConsumerV2`, the only changes to these classes are the substitutions of the `SyncronizedBuffer` class's name for the `Buffer` class's name. One similar highlighted substitution was made on line 8 of the application `PCThreadSync` presented in Figure 14.22, which is a modification of the application `PCThreadProblems` presented in Figure 14.15.

A typical output produced by the application `PCThreadSync` is given in Figure 14.23. As shown in the two-line summary at the bottom portion of the figure, the synchronization techniques incorporated into the shared buffer object have resulted in every produced data item (the integers 1 through 10) being consumed once and only once.

```
import java.util.Random;

public class ProducerV2 implements Runnable
{
  private SynhronizedBuffer sharedData;
  private Random delay = new Random();

  public ProducerV2(SynhronizedBuffer sharedData)
  {
    this.sharedData = sharedData;
  }

  public void run()
  {
    for(int i = 1; i <= 10; i++)
    {
      try
      { //simulate data processing
        Thread.sleep(delay.nextInt(10) + 1);
      }
      catch(InterruptedException e)
      {
      }
      sharedData.setData(i);
      System.out.println("Produced " + i);
    }
  }
}
```

Figure 14.20
The class ProducerV2 using a **`SynchronizedBuffer`** object.

```
import java.util.Random;

public class ConsumerV2 implements Runnable
{
```

```java
    private SynhronizedBuffer sharedData;
    private int[] timesConsumed = new int[10];
    private boolean[] consumedData = new boolean[10];

    public ConsumerV2(SynhronizedBuffer sharedData)
    {
      this.sharedData = sharedData;
    }

    public void run()
    {
      Random delay = new Random();
      int dataItem;

      for(int i = 1; i <= 10; i++)
      {
        try
        {
          Thread.sleep(delay.nextInt(10) + 1); //simulate data fetch
        }
        catch(InterruptedException e)
        {
        }
        dataItem = sharedData.getData();
        System.out.println("Consumed " + dataItem + " <---");

        //record consumed statistics
        consumedData[dataItem - 1] = true;
        timesConsumed[dataItem - 1]++;
      }
      outputConsumedSummary();
    }

    private void outputConsumedSummary() //output final statistics
    {
      try
      {
        Thread.sleep(5000);
      }
      catch(InterruptedException e)
      {
      }
      System.out.print("Consumed data: ");
      for(int i = 1; i <= 10; i++)
      {
        if(consumedData[i-1] == true)
        {
          System.out.print(" " + i);
        }
```

```
        }
        System.out.print("\nTimes consumed:");
        for(int i = 1; i <= 10; i++)
        {
          if(consumedData[i-1] == true)
          {
            System.out.print(" " + timesConsumed[i-1]);
          }
        }
    }
  }
```

Figure 14.21
The class ConsumerV2 using a **SynchronizedBuffer** object.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PCThreadSync
{
  public static void main(String[] args)
  {
    SynchronizedBuffer aBuffer = new SynchronizedBuffer();

    ProducerV2 producerThread = new Producer V2(Buffer);
    ConsumerV2 consumerThread = new Consumer V2(aBuffer);

    ExecutorService launcher = Executors.newCachedThreadPool();
    launcher.execute(producerThread);
    launcher.execute(consumerThread);

    launcher.shutdown();
  }
}
```

Figure 14.22
The application **PCThreadSync**.

```
Produced 1
Consumed 1 <---
Consumed 2 <---
Produced 2
Consumed 3 <---
Produced 3
Produced 4
Consumed 4 <---
Produced 5
Consumed 5 <---
Produced 6
```

```
Consumed 6 <---
Produced 7
Consumed 7 <---
Produced 8
Consumed 8 <---
Produced 9
Produced 10
Consumed 9 <---
Consumed 10 <---
Consumed data:  1  2  3  4  5  6  7  8  9  10
Times consumed: 1  1  1  1  1  1  1  1  1  1
```

Figure 14.23
The output produced by the application **`PCThreadSync`**.

One remaining issue concerning the output that is revealed in the first two highlighted lines output at the top portion of the Figure 14.23 should be discussed. Based on these outputs, it appears that data item 2 was consumed before it was produced. This was not the case.

What actually happened was that after data item 2 was produced by the producer thread's second invocation of the `setData` method (line 24 of Figure 14.20) and the `setData` method invoked `notifyAll` on line 28 of Figure 4.19, the consumer thread was brought from the waiting state to the runnable state. At that point, the producer thread's time quantum must have expired before line 25 of Figure 14.20 could execute and produce its output: *Produced 2*.

The next time quantum must have been awarded to the consumer thread, and its invocation of the `getData` method on line 28 of Figure 14.21 ran to completion consuming data item 2. The consumer thread's time quantum still had not expired, so line 29 of Figure 14.21 executed producing the puzzling output: *Consumed 2*.

When the producer thread was finally awarded a CPU time quantum, it continued where it left off when its previous time quantum had expired and executed line 25 of Figure 14.20, producing the output *Produced 2*. It was not the process of consuming and producing that was reversed; rather, it was the outputs that announce these events that were reversed. No data was lost. This same set of events occurred after data item 3 was produced, as evidenced by the third and fourth highlighted lines in Figure 14.23.

The remedy would be to perform the output inside of the synchronized methods `setData` and `getData`. This was purposely not done in order to reinforce the point that tasks (such as these output tasks) that need to be synchronized should be implemented inside synchronized methods and to illustrate the uncertainty of two sequential instructions (e.g., lines 24–25 of Figure 14.20) executing without interruption in a multithreaded application.

14.5.2 The API ArrayBlockingQueue Class

The API generic class `ArrayBlockingQueue` can be used as a producer consumer buffer. It is a synchronized thread safe implementation of a queue, which means that the solutions to the producer-consumer problems discussed in this chapter have been incorporated into the class.

A queue is a collection whose elements are maintained in a first-in-first-out order, which makes instances of this class well suited for use as a producer-consumer buffer. The first data item added to the buffer is the first item returned (and removed) from the buffer. The buffer stores references to objects because the class is a generic class. An instance of this class effectively performs all of the synchronization functionality of instances of the class `SynchronizedBuffer` shown in Figure 14.19.

The class's constructor is passed an integer parameter, which is the maximum number of elements that can be stored in the buffer at one time. The buffer is actually an array of reference variables, and this array-based queue is implemented as a circular queue. When used as a producer-consumer buffer, it is usually more efficient to increase the size of the buffer (the array) beyond one element, especially when the time between productions and consumptions can vary by a significant amount of time, because the producer and the consumer tasks do not have to alternate. That is, the producer could produce several items while the consumer is processing the first item.

The class's `put` method is invoked by the producer thread to add a data item, passed to it, to the rear of the queue. When the `put` method is passed a primitive value, the value is autoboxed before being added to the buffer. If the buffer is full when the `put` method is invoked, the method waits until an element becomes available (i.e., an item is consumed), and then the new item is added to the buffer.

The class's `take` method is invoked by the consumer thread to fetch a data item's address from the front of the queue and remove it from the buffer. If the buffer is empty when it is invoked, the method waits until an object is added to the buffer (i.e., an item is produced), and then its address is returned and deleted from the buffer.

The application `PCThreadSyncAPI`, shown in Figure 14.24, is a modified version of the synchronized producer-consumer multithreaded application presented in Figure 14.22. This version uses an instance of the API `ArrayBlockingQueue` as a data buffer. Figure 14.25 shows a typical output produced by its producer and consumer threads, which are now instances of the classes shown in Figures 14.26 and 14.27.

The changes to the application class shown in Figure 14.22 are highlighted in Figure 14.24. They include the change to declaration of the shared buffer object on lines 9 and 10, and the changes to the names of the producer and consumer classes on lines 11 and 12. The invocation of the constructor on line 10 is passed an `Integer` type argument because the buffer will store autoboxed integers. It is also passed a buffer size of one to make the application consistent with the application shown in Figure 14.22.

The producer and consumer classes shown in Figures 14.20 and 14.21 have been modified to reflect the change in the class of the shared buffer object. The modified versions of these classes are shown in Figures 14.26 and 14.27 and have been renamed `ProducerV3` and `ConsumerV3`. The type of the shared buffer object declared on line 6 of these classes has been changed to the `Array-BlockingQueue`, as has the type of their constructors' parameter (lines 9 and 10 in Figures 14.26 and 14.27, respectively).

Line 21 of the `ProducerV3` class invokes the `ArrayBlockingQueue` class's `put` method to add a data item to the buffer, and line 25 of the `ConsumerV3` class invokes the class's `take` method to fetch a data item from the buffer. The data item address returned from the `take` method is coerced into the `Integer` reference variable `dataItem` declared on line 18 of the consumer thread's class because the `take` method is generic and its returned type is `Object`.

The application does not include an implementation of a synchronized buffer class. That implementation was done for us by the author of the API `ArrayBlockingQueue` class.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ArrayBlockingQueue;

public class PCThreadSyncAPI
{
  public static void main(String[] args)
  {
    ArrayBlockingQueue <Integer> aBuffer;
    aBuffer = new ArrayBlockingQueue <Integer> (1);
    ProducerV3 producerThread = new ProducerV3(aBuffer);
    ConsumerV3 consumerThread = new ConsumerV3(aBuffer);

    ExecutorService launcher = Executors.newCachedThreadPool();
    launcher.execute(producerThread);
    launcher.execute(consumerThread);

    launcher.shutdown();
  }
}
```

Figure 14.24
The application **`PCThreadSyncAPI`**.

```
Produced 1
Consumed 1 <---
Produced 2
Consumed 2 <---
Consumed 3 <---
Produced 3
Produced 4
Consumed 4 <---
Produced 5
Consumed 5 <---
Produced 6
Consumed 6 <---
Produced 7
Consumed 7 <---
Produced 8
```

```
Consumed 8 <---
Consumed 9 <---
Produced 9
Produced 10
Consumed 10 <---
Consumed data:  1 2 3 4 5 6 7 8 9 10
Times consumed: 1 1 1 1 1 1 1 1 1 1
```

Figure 14.25
Output produced by the application **PCThreadSyncAPI**.

```java
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;

public class ProducerV3 implements Runnable
{
  ArrayBlockingQueue <Integer> sharedData;
  Random delay = new Random();

  public ProducerV3(ArrayBlockingQueue <Integer> sharedData)
  {
    this.sharedData = sharedData;
  }

  public void run()
  {
    for(int i = 1; i <= 10; i++)
    {
      try
      { //simulate data processing
        Thread.sleep(delay.nextInt(10) + 1);
        sharedData.put(i);
        System.out.println("Produced " + i);
      }
      catch(InterruptedException e)
      {
      }
    }
  }
}
```

Figure 14.26
The class **ProducerV3**.

```
import java.util.Random;
import java.util.concurrent.ArrayBlockingQueue;

public class ConsumerV3 implements Runnable
{
  ArrayBlockingQueue <Integer> sharedData;
  int[] timesConsumed = new int[10];
  boolean[] consumedData = new boolean[10];

  public  ConsumerV3(ArrayBlockingQueue <Integer> sharedData)
  {
    this.sharedData = sharedData;
  }

  public void run()
  {
    Random delay = new Random();
    Integer dataItem = 0;

    for(int i = 1; i <= 10; i++)
    {
      try
      {
        Thread.sleep(delay.nextInt(10) + 1); //simulate data fetch
        dataItem = (Integer) sharedData.take();
        System.out.println("Consumed " + dataItem + " <---");
      }
      catch(InterruptedException e)
      {
      }

      //record consumed statistics
      consumedData[dataItem - 1] = true;
      timesConsumed[dataItem - 1]++;
    }
    outputConsumedSummary();
  }

  private void outputConsumedSummary()
  {
    try
    {
      Thread.sleep(5000);
    }
    catch(InterruptedException e)
    {
    }
    System.out.print("Consumed data: ");
    for(int i = 1; i <= 10; i++)
```

```
      {
        if(consumedData[i-1] == true)
        {
          System.out.print(" " + i);
        }
      }
      System.out.print("\nTimes consumed:");
      for(int i = 1; i <= 10; i++)
      {
        if(consumedData[i-1] == true)
        {
          System.out.print(" " + timesConsumed[i-1]);
        }
      }
    }
  }
```

Figure 14.27
The class **ConsumerV3**.

14.6 THE SYNCHRONIZED STATEMENT

The synchronized statement is an alternative way of synchronizing a thread's access to a shared data item, which in this case, must be an object. The statement begins with the keyword **synchronized**, followed by a set of parentheses that enclose the thread's name for the shared data item, which is followed by a statement block as shown below:

```
synchronized (sharedObject)
{
    //one or more statements
}
```

In order for a thread to execute the statements contained in the code block, it must acquire a lock that Java associates with the shared object. If another thread has already acquired the lock, a thread that unsuccessfully attempted to acquire the lock enters the blocked state, as shown in the lower center portion of Figure 14.8. The lock is released when the synchronized statement block of the thread with the lock completes its execution. At that point, all threads that entered the blocked state, because they could not acquire the shared object's lock, return to the runnable state.

A thread does not have to acquire the lock to execute the methods of the shared data object when the invocation statement is not inside a synchronized statement's code block. This implies that the synchronized statement only restricts access to a shared data object's methods when at least two threads contain synchronized statements involving that object. In contrast, the *synchronized-method* construct discussed in the previous section prohibits all threads from executing any of the synchronized methods of an object until the threads acquire the object's lock. An understanding of this is fundamental to knowing how to use these two alternative synchronization features of the Java language and the restrictions they impose.

NOTE

Synchronized methods within a class cannot be executed concurrently, they are mutually exclusive.
Similarly, a synchronized statement on a shared object prevents threads that contain the same synchronized statement from concurrently executing the code block associated with their statement. The code blocks are mutually exclusive.

The application `SynchronizedStatement`, shown in Figure 14.28, demonstrates the use of the `synchronized` statement to produce an accurate count of the number of transactions made on an ATM system that has two ATM locations. Each ATM machine's transactions are processed by a separate thread in this multithreaded application. When the total number of transactions made on the ATM system reaches a maximum, which in the interest of brevity has been set to three, both ATM machines on the system are shut down.

The two `Runnable` objects, created on lines 9 and 10 of the application, are instances of the class `AtmTransaction` shown in Figure 14.29. When created, they are passed an instance of the buffer class `Counter` (Figure 14.30) declared on line 8 of the application. The objects share the `Counter` instance and use its methods to count the total number of transactions made on the ATM system.

Figure 14.31 shows two sets of outputs produced by the program. The output on the left side of the figure shows the two machines being shut down after three transactions have been made on the system. The output on the right side of the figure exhibits a producer-consumer problem: each ATM machine has performed three transactions for a total of six transactions. This output was produced when the synchronized statement on line 18 of Figure 14.29 was eliminated from the `AtmTransactions` class's `run` method.

The loop that begins on line 14 of Figure 14.29 simulates the threads' processing transactions at the ATM machines with which they are associated. The synchronized code block that begins on line 18 accomplishes the thread synchronization. When one of the ATM threads begins the code block's execution, the execution of the other thread in this class cannot proceed past line 18 until the execution of the code block, which ends on line 28, is complete.

Lines 25–28 increments the transaction `Counter` object shared by the threads. Line 26 moves the thread to the timed waiting state to simulate the thread's time quantum expiring between the fetching of the counter's value (line 25) and the incrementing and setting of it (line 27). When the thread's wait time interval expires and it is granted another CPU time quantum, the current transaction count is incremented, set, and then output to the system console (lines 27 and 28). While one ATM thread is in the timed waiting state, it still holds the lock on the shared data object, and the other thread cannot enter the synchronized statement's code block. This causes the sequence of fetching, incrementing, and setting the shared data item to be uninterruptible. The code to actually process a transaction (e.g., dispense funds) would replace the comment on line 34.

When the number of transactions performed on the ATM system reaches three, as determined by the `if` statement on line 21, the `break` statement (line 23) ends the loop, and the shutdown message is output by line 36. When the `synchronized` statement on line 19 is eliminated (i.e., commented out), the unsynchronized thread execution produces six transactions before the ATM system is shut down, as shown on the right side of Figure 14.31. The line-by-line unsynchronized execution sequence of the threads that produces the first two outputs (1 and 1) is shown in Figure 14.32. As the highlighted portions of the figure indicate, due to the simulated expiration of ATM 1's time quantum (line 26 in the top-left portion of the figure), ATM 2's thread is allowed to fetch the counter's value (line 25 in the top-right of the figure) before ATM 1's thread can write the incremented value to the shared counter (line 27 in the middle-left of the figure). The result is that each ATM increments a count of 0 and outputs a 1 (middle-left and bottom-right portions of the figure).

```
1    import java.util.concurrent.ExecutorService;
2    import java.util.concurrent.Executors;
3
4    public class SynchronizedStatement
5    {
6      public static void main(String[] args)
7      {
8        Counter shared = new Counter();
9        AtmTransaction ATM1 = new AtmTransaction(shared);
10       AtmTransaction ATM2 = new AtmTransaction(shared);
11
12       ExecutorService launcher = Executors.newCachedThreadPool();
13
14       launcher.execute(ATM1);
15       launcher.execute(ATM2);
16
17       launcher.shutdown();
18     }
19   }
```

Figure 14.28
The application **SynchronizedStatement**.

```
1    public class AtmTransaction implements Runnable
2    {
3      private Counter shared;
4
5      public AtmTransaction(Counter shared)
6      {
7        this.shared = shared;
8      }
9
10     public void run()
11     {
12       int count;
13
14       while(true)
15       {
```

```java
      //use of a synchronized statement
      try
      {
        synchronized(shared) //increments the transaction counter
        {
          if(shared.getCounter >= 3) //reached transaction limit
          {
            break;
          }
          count = shared.getCounter();
          Thread.sleep(10); //simulate end of time quantum
          shared.setCounter(count + 1);
          Counter.outputCounter();
        }
      }
      catch(InterruptedException e)
      {
      }
      //code to process a transaction would be coded here
    }
    System.out.println("An ATM is shutting down");
  }
}
```

Figure 14.29
The class **AtmTransaction**.

```java
public class Counter
{
  private static int counter = 0;

  public int getCounter()
  {
    return counter;
  }

  public void setCounter(int value)
  {
    counter = value;
  }

  public static void outputCounter()
  {
    System.out.println(counter);
  }
}
```

Figure 14.30
The class **Counter**.

With Synchronization	Without Synchronization

`1`
`2`
`3`
`An ATM shutting down`
`An ATM shutting down`
	`1`
	`1`
	`2`
	`2`
	`3`
	`An ATM shutting down`
	`3`
	`An ATM shutting down`

Figure 14.31
Output produced by the application **`AtmTransaction`**.

ATM 1 Thread	ATM 2 Thread
Lines 15-21	
Line 25 count = 0 = `shared.getCounter();`	
Line 26 Entered timed waiting state	
	Lines 15-21
	Line 25 count = 0 = `shared.getCounter();`
	Line 26 Entered Timed waiting state
Line 27 `shared.setCounter`(0 + 1);	
Lines 28 Output: 1	
Lines 15-21	
Line 25 count = 1 = `shared.getCounter();`	
Line 26 Enter timed waiting state	
	Line 27 `shared.setCounter`(0 + 1);
	Lines 28 Output: 1

Figure 14.32
SynchronizedStatement's thread-execution sequence when the synchronized statement is eliminated from the **`AtmTransaction`** class.

NOTE

Synchronized statements are used inside of classes that extend `Runnable` *to synchronize the invocations of a shared object's methods.*
Synchronized methods are used inside a shared object's class to synchronize invocations to the shared object's methods.

14.7 CHAPTER SUMMARY

Programs can be divided into a two or more independently executing parts called threads. Threads are instances of a class that implements the interface `Runnable` or a class that extends the class `Thread` (which implements the interface `Runnable`). These objects are typically created, and their execution is initiated from within the program's `main` method. Once initiated, their execution begins with the first executable statement in their `run` method, whose signature is defined in the interface `Runnable`.

During the lifetime of a thread, which is the time between when the thread is created and when it is terminated, it exists in one of six Java states named new, runnable, blocked, waiting, timed waiting, and terminated. When a thread object is created, it enters the new state, and when it completes its execution, it enters the terminated state. The remainder of its lifecycle is spent in the other four states. Threads that are in the runnable state can be assigned a quantum of execution time by the operating system, and the algorithm used to assign threads in the runnable state a quantum of commuting time is platform dependent. Events, such as performing input or output, can cause threads to transition out of and back into the runnable state, as can invocations of the methods `notify`, `notifyAll`, `wait`, and `sleep`.

Threads can share data and communicate with each other via the data members of an object, which is referred to as a shared data buffer object. The address of the buffer object is passed to the threads' constructor when they are created, and the threads access the data using the buffer class's `set` and `get` methods or equivalent methods. When the shared data is writeable, such as in the producer-consumer problem, a variety of problems can develop that do not occur in single threaded applications. The Java synchronized method construct and Java's `synchronized` statement can be used to avoid these problems.

When the `set` and `get` methods are coded as synchronized methods, and a thread attempts to initiate their execution before a previously initiated execution of them is completed, the thread is removed from the runnable state and placed in the waiting state. The thread is returned to the runnable state when the `set` or `get` method in execution issues an invocation to the `notify` or `notifyAll` method at the end of its execution. When the synchronized statement is used, the threads sharing the buffer object place their invocations of the `set` and `get` methods inside the synchronized statement's code block. When a thread's execution path attempts to execute the statements within the code block while another thread's synchronized statement's code block is in execution, the thread is moved from the runnable state to the blocked state. It is returned to the runnable state when the processing performed by the statements in the code block of the synchronized statement in execution is completed.

An instance of an API class that is designated to be thread safe can be used as a data buffer object to share data among threads in a safe, problem-free way. The generic class `ArrayBlockingQueue`, which is part of the collections framework, is thread safe. When a buffer object in this class is shared by two or more producer and consumer threads the producer threads are blocked from overwriting objects in the buffer that have not been consumed, and the consumer threads are blocked from fetching objects from the buffer that have not been produced or have been previously consumed. In addition, the buffer can be specified to hold more than one item, which can improve the performance of a producer-consumer application.

Knowledge Exercises

1. True or False:

a) All Java programs contain at least one thread.

b) Programs with multiple threads must be run on a system that has more than one CPU.
c) The scheduling algorithm used to assign a CPU to a thread is platform dependent.
d) Two threads can never be in execution at the same time.
e) A thread's class must implement the Java API class `Runnable`.
f) The method `start` can be used to initiate the execution of a thread.
g) A thread's class does not have to contain a method named `run`.
h) Once a thread begins its execution, it always continues to execute until it completes its execution.

2. If you wanted a thread to output the message *Thread1 is executing* as soon as it begins executing, where would you code the output statement?

3. Give the states a Java thread can be in during its lifetime.

4. In which state must a Java thread be for it to receive a quantum of execution time?

5. Give the method(s) invoked to place a thread in the waiting state.

6. Give the method(s) invoked to place a thread in the timed waiting state.

7. Give two ways a thread could enter the waiting state.

8. Give two ways a thread could enter the blocked state.

9. True or false:
a) A thread can place itself into the waiting state.
b) A thread can remain in the waiting state indefinitely.
c) When a thread leaves the blocked state, it enters the waiting state.
d) A thread can be in two states at the same time.
e) A thread in the terminated state can return to the runnable state.
f) After a thread's execution is initiated, it enters the new state.
g) Once a thread leaves the runnable state, it cannot return to that state.
h) The invocation `wait(3)` places a thread in the timed waiting state for three seconds.

10. Give the code to:
a) Create and initiate the execution of a thread whose class `Output` extends the class `Thread`.
b) Create and initiate the execution of a thread whose class `Input` implements the interface `Runnable`.
c) Accomplish Exercise 9c using an executor service.

11. Explain the producer and consumer problem and some of the solutions to it.

12. Define the term thread safe in the context in which it is used in the API documentation.

13. Give the two features of Java that can be used to synchronize the access to a shared buffer object.

14. True or false:

a) A producer thread reads data from a shared buffer.

b) If threads sharing a data item only fetch its value, there is no need to synchronize access to it.

c) All of the API classes are thread safe.

d) The API class `ArrayBlockingQueue` is thread safe.

e) When a class that is thread safe is used to share data among threads, deadlock cannot occur.

Programming Exercises

In the following exercises, do not use an API class to share data between threads unless explicitly told to do so.

1. Give a thread safe invocation of the `getY` method on the shared buffer object `xyLocation` that is an instance of a non-thread safe class.

2. A thread safe class can be used to share its string data member `title` among concurrent threads. Give the code of the class's `setTitle` method.

3. Write a program that creates and launches a thread that outputs the string *Happy Birthday Nadia* a given number of times to the system console, one output per line. The number of times to perform the output will be input by the program user via an input dialog box and passed to the thread class's one-parameter constructor.

4. Write a program that creates and launches two threads that output their names a given number of times to the system console, one output per line. Their names and the number of times to output their names will be input by the program user via a message dialog box and passed to the thread class's two-parameter constructor. The two threads should be instances of the same class and launched after all the user I/O is complete. Use the program to demonstrate, via the program's output, that threads share a CPU's computing time.

5. Repeat Exercise 2, but this time, the threads should be instances of two different classes, and the thread that produces the most output should complete its output before the other thread begins its output, regardless of the order in which the threads are launched. Verify the correctness of your program when one of the threads produces a large amount of output by reversing the order in which the threads are launched.

6. Write a program that computes and outputs two terms of the Fibonacci sequence whose term numbers are input by the program user, each term being calculated concurrently by a separate thread. After accepting the two inputs, the application will create and launch the threads. The thread class should invoke the recursive method given below to perform the calculation. Use the program to discover a set of inputs that causes the first thread launched to complete its execution first and a set of inputs that causes the second thread launched to complete its execution first.

```
public static long fibonacci(long n)
{
  if(n==1 || n==2)
```

```
        {
          return 1;
        }
        else
        {
          return(fib(n-1) + fib(n-2));
        }
      }
```

7. Write a program that repeatedly asks the user to enter an integer via an input dialog box and outputs the integer to the system console, until the user enters -1. The input and output should be performed by two separate threads. After each output, the output thread should enter the waiting state for a random amount of time between 1 and 5 seconds. Examine the output to be sure that every number input is output when you enter the inputs as rapidly as possible.

8. Write a program that repeatedly computes and outputs the nth term of the Fibonacci sequence, with the term number being input by the program's user, until the user enters -1. The main method will launch two threads: one that performs the input and another that calculates and outputs the value of the term to the system console. Examine the output to make sure that there is an output for every input when the term numbers are input as rapidly as possible and the term numbers are in the range of 30–50. The calculation/output thread should invoke the method given in Exercise 6 to compute the value of the Fibonacci term. Perform the synchronization using synchronized methods in the buffer class.

9. Repeat Exercise 8 using synchronized statements.

10. Repeat Exercise 8 using an instance of the API `ArrayBlockingQueue` class as a shared data buffer. The buffer should only hold one data item. Which approach is less work and therefore more efficient?

11. Repeat Exercise 10 using an instance of the API `ArrayBlockingQueue` class that can hold four shared data items. Discover the range of Fibonacci term numbers for which this sized buffer noticeably improves the program's performance and explain why this is the case.

Enrichment

1. Investigate classical synchronization problems such as the Dining Philosophers problem and the Readers-Writers problem.

2. Explore other thread synchronization techniques used in hardware or software.

3. Find other examples of synchronization in everyday problems, such as accessing a shared database.

4. Look for other synchronization problems such as the Sleeping Barber problem and explain how these are similar to problems presented in this chapter.

References

Silberschatz, Abraham, et al. *Operating System Concepts*, 9th Ed. New York: John Wiley and Sons, 2013.

APPENDIX **A**

DESCRIPTION OF THE GAME ENVIRONMENT

A.1 OVERVIEW OF THE GAME ENVIRONMENT

The game environment is comprised of the interface `Drawable` and the two classes `DrawableAdapter` and `GameBoard`. To use the game environment in an application, the interface and these two classes must be included as part of the application (see Appendix B).

Figure A.1 shows a Java application that displays the game environment's window shown in Figure A.2. It assumes the game environment package `edu.sjcny.gpv1` has been added to the system's CLASSPATH variable. If the alternate approach described in Appendix B and in the *IDE Tools* subfolder contained on the book's DVD, which does not require a change in the system's CLASSPATH variable, was used to incorporate the game environment into the application's project, the import statement may not be necessary.

```
1  import edu.sjcny.gpv1.*; //May not be necessary
2  public class GameWindowDemo extends DrawableAdapter
3  {
4    static GameWindowDemo ga = new GameWindowDemo();
5    static GameBoard gb = new GameBoard(ga, "The Game's Title");
6
7    public static void main(String[] args)
8    {
9       showGameBoard(gb);
10   }
11 }
```

Figure A.1
The application **`GameWindowDemo`** that displays the game environment window.

As shown on line 2 of Figure A.1, game programs that use the game environment must extend the class `DrawableAdapter` and declare a static class level instance of the application's class using the default (no-parameter) constructor, as shown on line 4. Then, an instance of the class `GameBoard` is declared, passing the constructor the class level instance of the application's class and the title of the game (line 5). Finally, the `showGameBoard` method in the `DrawableAdapter` class is invoked from within the `main` method (line 9) and passed the `GameBoard` object declared on line 5. This method displays the application's window shown in Figure A.2.

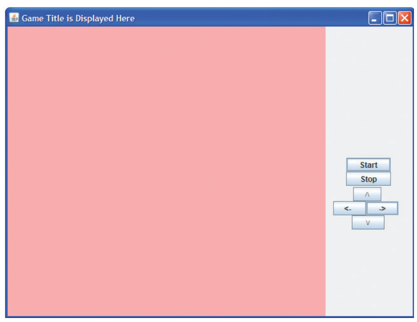


Figure A.2
A game application's window.

The Game Window

As shown in Figure A.2, a game application's window contains six buttons on its right side. The large pink panel to the left of the buttons is called the game board. Game piece objects are displayed on the game board. The color of the game board can be changed by invoking the API `Component` class's `setBackground` method on the `GameBoard` object (declared on line 5 of Figure A.1) and passing it the new board color (an instance of an API `Color` class object).

Timers and Timer Methods

In addition to the six buttons, a `GameBoard` object has three timers associated with it. These begin ticking when the game's player clicks the game window's Start button, and they stop ticking when the game window's Stop button is clicked. The `GameBoard` class contains methods (subprograms) that the programmer can invoke to stop and start a timer, and a method to change the rate at which a timer ticks. These methods are invoked on the `GameBoard` object declared on line 5 of Figure A.1. Section A.2.1 gives the signatures and a description of each of these methods. By default, the tick rates of the timers (named 1, 2, and 3) are once every second, once every half second, and once every quarter second, respectively.

Each timer has a call back method, or subprogram, associated with it, which the game environment invokes every time the timer ticks. The names of the methods, which are described in Section A.2.3, are `timer1`, `timer2`, and `timer3`. The class `DrawableAdapter` contains empty implementations of the methods. If the programmer includes (overrides) these methods in a game program, the game environment will invoke (or "call back") the programmer's versions of the methods after every tick of the timers. Java code placed inside of these methods can be used to keep track of a game's time and to animate objects on the game board.

Game Player Action Methods

There are eight other call back methods in the game environment, which are described in Section A.2.3. Seven of these are invoked by the game environment when the game player performs input actions common to most games. Four of these are associated with the game window's left, right, up, and down buttons, one is associated with the keyboard, and two are associated with the mouse. Empty implementations of the methods are coded in the class `DrawableAdapter`. If the programmer includes (overrides) these methods in a game program, the game environment invokes (calls back) the programmer's overridden version of the methods every time the game player clicks a directional button, presses a keyboard key, or drags or clicks the mouse on the game board.

The eighth call back method in this group of methods is associated with redrawing the game window. It is called by the game environment every time the game window needs to be redrawn (e.g., is minimized and then restored) and every time the seven game player input action call back methods or the three timer call back methods complete their execution. An empty implementation of this method is also included in `DrawableAdapter` class.

Game Window Coordinate System

The game window has a Cartesian coordinate system associated with it as shown in Figure A.3. The coordinate system's origin is in the upper left corner of the game window, with the positive x direction to the right and the positive y direction downward. The height and width of the top and left borders of the window place the upper left corner of the game board at (5, 30). The game window can be sized by using the last two parameters of the game board's four-parameter constructor to specify the coordinates of the lower right corner of the game board, which defaults to (500, 500).

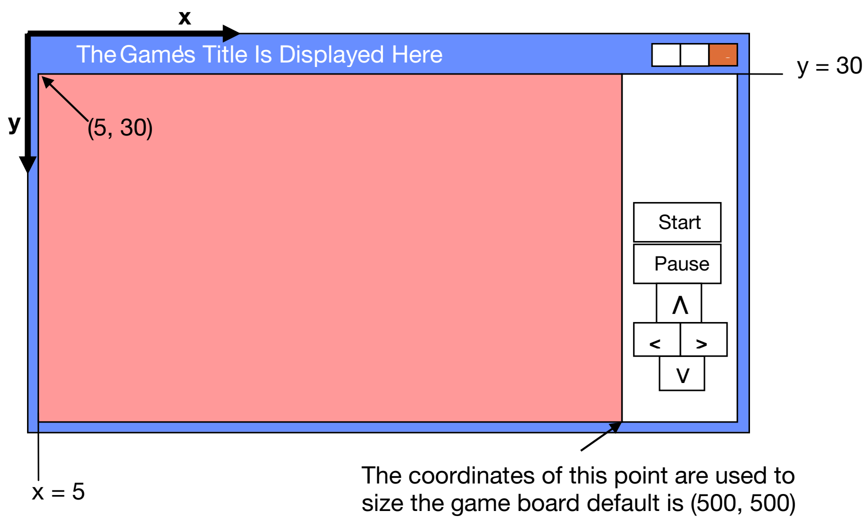


Figure A.3
The game environment coordinate system.

A.2 DESCRIPTION OF THE GAME ENVIRONMENT'S CLASS AND INTERFACE

The game environment is comprised of two classes, named `GameBoard` and `DrawableAdapter`, and one interface named `Drawable`.

A.2.1 The GameBoard Class

The class `GameBoard` contains two constructors. One creates a default-sized game board with a programmer specified title, and the other adds the ability to create a game board and specify its size. A game application must construct a static `GameBoard` object and pass it a static instance of the application's class (see lines 4 and 5 of Figure A.1). Assuming the static instance of the application class was named `ga`, the following line of code creates the `GameBoard` instance `gb` with a default game board size, 500 x 500 pixels:

```
static GameBoard gb = new GameBoard(ga, "The Game's Title");
```

The `GameBoard` object can be used within the game application class to invoke the class's other three methods that are used to set the time increment of any of the `GameBoard` object's three timers and to start and stop these timers. The following invocation stops timer 2:

```
gb.stopTimer(2);
```

Constructors

```
public GameBoard(Object app, String windowTitle)
```

This method constructs a `GameBoard` object, defaulting to the coordinates of the lower right corner of the game board to (500, 500).

Parameters:

`app` is an instance of the application's class.
`windowTitle` will be displayed as the title of the application's window in its title bar.

```
public GameBoard(Object app, String windowTitle,
                 int xMaxValue, int yMaxValue)
```

This method constructs a `GameBoard` object whose lower-right corner is specified by the last two arguments passed to it.

Parameters:

`app` is an instance of the application's class.
`windowTitle` will be displayed as the title of the application's window in its title bar.
`xMaxValue` is the x-pixel coordinate of the lower-right corner of the game board.
`yMaxValue` is the y-pixel coordinate of the lower-right corner of the game board.

Methods

```
public void setTimerInterval(int timerNumber, int interval)
```

This method sets the interval of one of the game environment's three timers. The default increments for the timers are 1000ms (1 second) for timer 1, 500ms (1/2 second) for timer 2, and 250 ms (1/4 second) for timer 3.

Parameters:

`timerNumber` is the number of the timer (1, 2, or 3) whose interval is being set.
`interval` is the time between ticks of the timer in milliseconds (e.g., 1000 = 1 second).

```
public void startTimer(int timerNumber)
```

This method starts the timer whose number (1, 2, or 3) is passed to it. While the timer is ticking, the timer's call back method (`timer1`, `timer2`, or `timer3`) will be executed on each subsequent tick of the timer. Ticking is stopped when the game player clicks the game board's Pause button or when the `GameBoard` class's `stopTimer` method is invoked. Ticking is restarted when the Start button is clicked or this method is reinvoked.

Parameters:

`timerNumber` designates the timer to be started: 1, 2, or 3.

```
public void stopTimer(int timerNumber)
```

This method stops the timer whose number (1, 2, or 3) is passed to it. After this method is invoked, clicking the game board's Start button will *not* restart the timer. The timer's call back method (`timer1`, `timer2`, or `timer3`) will not be executed until the timer is started again via an invocation of the `startTimer` method, which will also reactivate the game board's Start button.

Parameters:

`timerNumber` designates the timer to be stopped: 1, 2, or 3.

A.2.2 The DrawableAdaper Class

A game program's class must extend the class `DrawableAdaper` (as on line 2 of Figure A.1). It provides empty implementations of the eleven call back methods defined in the game package interface `Drawable`. A description of these eleven methods is given in Section A.2.3. In addition, it provides a method that displays an instance of a `GameBoard` object passed to it (line 9 of Figure A.1).

Methods of the Class DrawableAdapter

```
public static void showGameBoard(GameBoard gb)
```

This method displays the game program's window. Normally, it is invoked as the last statement in the game program's `main` method.

Parameters:

`gb` is the application's `GameBoard` object.

A.2.3 The Interface Drawable

The interface `Drawable` defines eleven call back methods invoked by the game environment. They are coded (as required) in the game application's class (the class that contains the method `main`). They are used to service various actions by the game's player (e.g., a mouse click or drag, a keystroke, or a button click), and to perform processing such as animation every time a game environment's timer ticks.

The call back method `draw` is invoked by the game environment when the game window has to be redrawn (e.g., it is dragged to a new location) and after any of the other ten call back methods complete their execution.

Call Back Methods

`public void draw(Graphics g)`

This method is invoked when the game application window is initially displayed or needs to be redisplayed and each time one of the other ten call back methods complete their execution.

Parameters: `g` is an instance of the API class `Graphics` attached to the game board, which is passed into this method when it is invoked by the game environment. It can be used to draw two-dimensional shapes on the game board by invoking the methods in the `Graphics` class.

`public void timer1()`

This method is invoked every time timer 1 ticks. The timer ticking can be started or stopped by invoking the `GameBoard` class's `startTimer` and `stopTimer` methods, respectively. If the timer is ticking, it is stopped whenever the Stop button in the game's window is clicked and restarted whenever the Start button in the game's window is clicked.

`public void timer2()`

This method is invoked every time timer 2 ticks. The timer ticking can be started or stopped by invoking the `GameBoard` class's `startTimer` and `stopTimer` methods, respectively. If the timer is ticking, it is stopped whenever the Stop button in the game's window is clicked and restarted whenever the Start button in the game's window is clicked.

`public void timer3()`

This method is invoked every time timer 3 ticks. The timer ticking can be started or stopped by invoking the `GameBoard` class's `startTimer` and `stopTimer` methods, respectively. If the timer is ticking, it is stopped whenever the Stop button in the game's window is clicked and restarted whenever the Start button in the game's window is clicked.

`public void leftButton()`

This method is invoked whenever the Left button in the game's window is clicked.

`public void rightButton()`

This method is invoked whenever the Right button in the game's window is clicked.

`public void upButton()`

This method is invoked whenever the Up button in the game's window is clicked.

`public void downButton()`

This method is invoked whenever the Down button in the game's window is clicked.

`public void keyStruck(char key)`

This method is invoked whenever a key on the keyboard is struck. If the key is held down, the method is continually invoked until the key is released.

Parameters: `key` contains the upper case version of the character that was struck. The cursor control keys return 'L', 'R', 'U' or 'D' when the left, right, up, or down arrows are struck.

`public void mouseClicked(int x, int y, int buttonPressed)`

This method is invoked whenever a mouse button is clicked.

Parameters: `x` and `y` are the game board coordinates of the mouse cursor location at the time the mouse was clicked.

`buttonPressed` contains a 1 if the left mouse button was clicked or a 3 if the right mouse button was clicked.

`public void mouseDragged(int x, int y)`

This method is continually invoked while the mouse is being dragged.

Parameters: `x` and `y` are the game board coordinates of the mouse cursor location at the time the method is invoked.

Appendix B

Using the Game Environment Package

The game environment can be easily used within the Eclipse, NetBeans, and JCreator IDEs without changing the operating system's CLASSPATH variable by following the IDE-specific directions listed below. Alternately, the package `edu.sjcny.gbv1`, which is in the `Package` subfolder of the `Game Environment\Class, Package and JAR file` folder on the DVD that accompanies this text can be stored on your system and added to its CLASSPATH variable. Then, the following import statement can be used to incorporate the game environment into a game application:

```
import edu.sjcny.gpv1.*;
```

NON-CLASSPATH ALTERING IDE-SPECIFIC INSTRUCTIONS

Eclipse IDE

Method 1: Import the Eclipse project template

1. Create a folder and bring up Eclipse into that folder.
2. Import the project `EclipseGameTemplate7` into the folder.
   - Click File - Import – General - Existing Projects into Workspace – Next
   - Browse to the DVD folder:
     `Game Environment\IDE Specific Tools\Eclipse\Workspace`
     and click the `EclipseGameTemplate7` template folder, then click OK
   - Check the box next to *Copy Projects Into Workspace*, then click Finish
3. Open the project `EclipseGameTemplate7` and add the program-specific code to it.

Method 2: Add the game environment JAR file or its classes to a new Eclipse project

Either the JAR file `gameEnvironment.jar` contained in the folder `GameJAR` or the classes contained in the `GameClasses` folder can be added to any existing Eclipse project and its build path. Both of these folders are in the `Game Environment\IDE Specific Tools\Eclipse` subfolder on the DVD that accompanies this textbook. To add them to an existing Eclipse project's build path:

1. Launch Eclipse in the existing project's workspace
2. Locate and copy the folder `GameJAR` or `GameClasses`
3. Right click the project node in Eclipse's Package Explorer view pane, then click Paste
4. Right click the project node in the Package Explorer view pane, then click Properties - Java Build Path - Libraries
   (a) To add the `gameEnvironment.jar` file, click "Add JAR's…" and locate and check the `gameEnvironment.jar` JAR file, click OK, click OK

(b) To add the `GameClasses` folder, click "Add Class Folder" and locate and check the `GameClasses` folder, click OK, click OK

NetBeans IDE

1. Create a folder with a name relevant to the program being developed
2. Copy the NetBeans project `NBGameTemplate7` located in the `Game Environment\IDE Specific Tools\NetBeans` subfolder on the DVD that accompanies this textbook and paste it into the folder created in Step 1
3. Open the project `NBGameTemplate7` and add the program-specific code to it

JCreator IDE

Method 1

1. Create a folder with a name relevant to the program being developed
2. Copy the JCreator project `JCGameTemplate7` located in the `Game Environment\IDE Tools\JCreator` subfolder on the DVD that accompanies this textbook and paste it into the folder created in Step 1
3. Open the project `JCGameTemplate7` and add the program-specific code to it

Method 2

1. Create a JCreator project
2. Copy and paste the folder `edu` (i.e., the package `edu.sjcny.gpv1`, contained in the `Game Environment\IDE Tools\JCreator` subfolder on the DVD that accompanies this book, into the project's class folder
3. Include the following import statement in the application:

```
import edu.sjcny.gpv1.*;
```

Appendix C

ASCII Table

Decimal	Octal	Hex	Binary	Char	Description
000	000	000	00000000	NUL	(null)
001	001	001	00000001	SOH	(start of Heading)
002	002	002	00000010	STX	(start of text)
003	003	003	00000011	ETX	(end of text)
004	004	004	00000100	EOT	(end of transmission)
005	005	005	00000101	ENQ	(enquiry)
006	006	006	00000110	ACK	(acknowledge)
007	007	007	00000111	BEL	(audible bell)
008	010	008	00001000	BS	(backspace)
009	011	009	00001001	HT	(horizontal tab)
010	012	00A	00001010	LF	(line feed, new line)
011	013	00B	00001011	VT	(vertical tab)
012	014	00C	00001100	FF	(form feed)
013	015	00D	00001101	CR	(carriage return)
014	016	00E	00001110	SO	(shift out)
015	017	00F	00001111	SI	(shift in)
016	020	010	00010000	DLE	(data link escape)
017	021	011	00010001	DC1	(device control 1)
018	022	012	00010010	DC2	(device control 2)
019	023	013	00010011	DC3	(device control 3)
020	024	014	00010100	DC4	(device control 4)
021	025	015	00010101	NAK	(negative acknowledge)
022	026	016	00010110	SYN	(synchronous idle)
023	027	017	00010111	ETB	(end of trans. block)
024	030	018	00011000	CAN	(cancel)
025	031	019	00011001	EM	(end of medium)
026	032	01A	00011010	SUB	(substitute)
027	033	01B	00011011	ESC	(escape)
028	034	01C	00011100	FS	(file separator)
029	035	01D	00011101	GS	(group separator)
030	036	01E	00011110	RS	(record separator)
031	037	01F	00011111	US	(unit separator)
032	040	020	00100000	SP	(space)
033	041	021	00100001	!	

(Contd.)

Decimal	Octal	Hex	Binary	Char	Description
034	042	022	00100010	"	
035	043	023	00100011	#	
036	044	024	00100100	$	
037	045	025	00100101	%	
038	046	026	00100110	&	
039	047	027	00100111	`	
040	050	028	00101000	(	
041	051	029	00101001	)	
042	052	02A	00101010	*	
043	053	02B	00101011	+	
044	054	02C	00101100	,	
045	055	02D	00101101	-	
046	056	02E	00101110	.	
047	057	02F	00101111	/	
048	060	030	00110000	0	
049	061	031	00110001	1	
050	062	032	00110010	2	
051	063	033	00110011	3	
052	064	034	00110100	4	
053	065	035	00110101	5	
054	066	036	00110110	6	
055	067	037	00110111	7	
056	070	038	00111000	8	
057	071	039	00111001	9	
058	072	03A	00111010	:	
059	073	03B	00111011	;	
060	074	03C	00111100	<	
061	075	03D	00111101	=	
062	076	03E	00111110	>	
063	077	03F	00111111	?	
064	100	040	01000000	@	
065	101	041	01000001	A	
066	102	042	01000010	B	
067	103	043	01000011	C	
068	104	044	01000100	D	
069	105	045	01000101	E	
070	106	046	01000110	F	
071	107	047	01000111	G	
072	110	048	01001000	H	
073	111	049	01001001	I	
074	112	04A	01001010	J	

Decimal	Octal	Hex	Binary	Char	Description
075	113	04B	01001011	K	
076	114	04C	01001100	L	
077	115	04D	01001101	M	
078	116	04E	01001110	N	
079	117	04F	01001111	O	
080	120	050	01010000	P	
081	121	051	01010001	Q	
082	122	052	01010010	R	
083	123	053	01010011	S	
084	124	054	01010100	T	
085	125	055	01010101	U	
086	126	056	01010110	V	
087	127	057	01010111	W	
088	130	058	01011000	X	
089	131	059	01011001	Y	
090	132	05A	01011010	Z	
091	133	05B	01011011	[	
092	134	05C	01011100	\	
093	135	05D	01011101	]	
094	136	05E	01011110	^	(caret)
095	137	05F	01011111	_	(underscore)
096	140	060	01100000	`` ` ``	
097	141	061	01100001	a	
098	142	062	01100010	b	
099	143	063	01100011	c	
100	144	064	01100100	d	
101	145	065	01100101	e	
102	146	066	01100110	f	
103	147	067	01100111	g	
104	150	068	01101000	h	
105	151	069	01101001	i	
106	152	06A	01101010	j	
107	153	06B	01101011	k	
108	154	06C	01101100	l	
109	155	06D	01101101	m	
110	156	06E	01101110	n	
111	157	06F	01101111	o	
112	160	070	01110000	p	
113	161	071	01110001	q	
114	162	072	01110010	r	
115	163	073	01110011	s	

(Contd.)

Decimal	Octal	Hex	Binary	Char	Description
116	164	074	01110100	t	
117	165	075	01110101	u	
118	166	076	01110110	v	
119	167	077	01110111	w	
120	170	078	01111000	x	
121	171	079	01111001	y	
122	172	07A	01111010	z	
123	173	07B	01111011	{	
124	174	07C	01111100	|	(vertical bar)
125	175	07D	01111101	}	
126	176	07E	01111110	~	(tilde)
127	177	07F	01111111	DEL	(delete)

Appendix D

Java Key Words

Java Keywords				
abstract	default	if	private	this
assert[2]	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	return	transient
byte	enum[3]	int	short	try
case	extends	interface	static	void
catch	final	long	strictfp[1]	volatile
char	finally	native	super	while
class	float	new	switch	
continue	for	package	synchronized	

1: added in version 1.2
2: added in version 1.4
3: added in version 5.0

Appendix E

Java Operators and Their Relative Precedence

1 is highest precedence

Operator	Description	Precedence	Operator	Description	Precedence
postfix operators		1	*equality operators*		7
++	postfix increment		==	is equal to	
--	postfix decrement		!=	is not equal to	
unary operators		2	*bitwise* AND		8
++	prefix increment		&	bitwise AND	
--	prefix decrement		*bitwise exclusive* OR		9
+	leading plus		^	exclusive OR	
-	leading minus		*bitwise inclusive* OR		10
!	logical not		|	inclusive OR	
~	Bitwise complement		*logical* AND		11
multiplicative operators		3	&&	conditional AND	
*	multiplication		*logical* OR		12
/	division		||	conditional OR	
%	remainder		*ternary*		13
additive operators		4	?:	conditional	
+	addition		*assignment*		14
-	subtraction		=	assignment	
shift operators		5	+=	addition assignment	
<<	shift left		-=	subtraction assignment	
>>	shift right		*=	multiplication assignment	
>>>	unsigned shift right		/=	division assignment	
relational operators		6	%=	remainder assignment	
<	less than		&=	bitwise AND assignment	
<=	less than or equal to		^=	bitwise exclusive OR assignment	
>	greater than		|=	bitwise inclusive OR assignment	
>=	greater than or equal to		<<=	bitwise left shift assignment	
instanceof	class comparator		>>=	bitwise right shift assignment	
			>>>=	bitwise unsigned right shift assign	

APPENDIX F

Using the Game

Glossary of Programming Terms

Abstract class A class that includes the keyword abstract in its signature and cannot be instantiated; it is used during the design process to collect data members and methods common to several classes

Aggregated class A class that contains at least one data member that references an object

Aggregation The concept of referencing objects from a class's data members

Algorithm A step-by-step solution to solving a problem or performing a task that a computer system can execute

Applet A Java program that runs from within another program, which is typically a Web browser; it has restrictions placed on its instruction set consistent with this execution mode's need for enforced security

Applet container program The program, typically a Web browser, within which an applet runs; the container invokes the methods that are part of the applet's lifecycle

Applet lifecycle The period of time that begins when an applet's execution is initiated and ends when it is terminated; during this time period, the applet's container program invokes the applet methods `init`, `start`, `paint`, `stop`, and `destroy` to manage its execution

Application Programming Interface (API) A collection of packages containing interfaces and implementations of classes and data structures that can easily be incorporated into a Java program

Application software All non-operating system software, typically for use by human users

Argument A value passed to a method when it is invoked

Argument list A sequence of argument names separated by commas enclosed in a set of parentheses

Array An ordered collection of primitive or reference variables stored inside an object, which are sequentially associated with an integer beginning with zero; arrays are an implementation of the mathematical concept of subscripted variables

Array of objects An array of reference variables that contains the addresses of a set of instances of the same class

ASCII Table A specific tabulation of characters and control characters and the bit patterns used to represent them

Assignment The act of changing the contents of a variable

Atomic components Graphical user interface (GUI) components that cannot contain other components, such as text fields and buttons; most of the program user's interactions are with these components

Autoboxing A context-sensitive feature of Java in which primitive literals or variables are replaced with instances of wrapper classes that contain their values

Base case Part of the methodology of formulating recursive algorithms, which is a known or trivial solution to the problem

Base class A class that is inherited from, also known as a parent or super class

Binary numbers A number system based on two, as opposed to the decimal system, which is based on ten

Bit A single unit of storage that can assume two states, which are referred to as off and on, or zero and one, or false and true

Boolean expression An expression involving relational and logic operators that evaluates to true or false

Buffer Memory used to temporarily store data during program execution

Byte A set of eight contiguous (adjacent) bits, often used to represent a single character in the Modern Latin (English) alphabet

Byte codes The translation of a program produced by the Java language translator into intermediate code

Central processing unit (CPU) Electronic circuitry that interprets and executes instructions; the CPU can perform arithmetic and logic operations, has the ability to skip or re-execute instructions based on the truth value of a logic operation, and contains a limited amount of storage called registers

Chain inheritance When the parent class of a class extends another class

Child class A class that inherits from (extends) another class; also known as a sub- or derived class

Class A collection of variables and methods; a blueprint for an object

Class-level variable A variable defined within a class but outside of a method's code block

Cloning an object Creating a new instance of a class and (deep) copying the values of all of the data members of an existing instance of the class into the new instance

Code block A set of instructions enclosed with a set of open and close braces, { }

Collection A data structure that is accessed without specifying a key

Collections Framework of the API Part of the API that contains generically implemented data structures, methods that perform common operations on data elements, and a set of associated interfaces

Computer system A set of electronic circuits, mechanical devices and enclosures, and instructions that these devices execute to perform a task

Concatenation The act of appending one string to another

Concurrency Executing several programs, or several parts of a program, at the same time

Constructor A method in a class that is used to create an instance of a class and return its address; its name is the same as the class's name

Consumer A process that expends data

Content pane The portion of a widow or other top-level container that holds the visible components added to the container

Control of flow statement A statement that overrides the default sequential execution path of a program, such as a decision statement, a repetition (loop) statement, or a subprogram invocation

Counting algorithm An algorithm that counts by adding an increment to, or subtracting a counting increment from, the current value of a counter

Data members The variables defined within a class

Data structure An organization of data within memory to facilitate its processing from a speed and memory requirements viewpoint

Deep comparison of two objects Comparing the values of one or more of the data members of an object to the corresponding data members of another instance of the class

Deep copy of an object Copying the values of one or more of the data members of an instance of a class into the corresponding data members of another instance of the class

Derived class A class that inherits from (extends) another class; also known as a child or subclass.

Deserializing objects The act of reassembling objects after they are read from a disk file

Dialog box A predefined pop-up graphical interface used to pause a program's execution until the program user acknowledges a message or performs an input

Divide and conquer Expressing or defining a complicated entity as a set of less complicated entities; for example, expressing the solution to a complex problem as the solutions to a set of simpler problems, or defining the data members of a complex class to be instances of less complicated classes

Dynamic binding Delaying the process of locating an invoked method until runtime

Dynamic programming A programming technique aimed at reducing execution time, which avoids repetitive processing by saving and then reusing prior processing results

Element One of the variables contained in an array or one object contained in a data structure

Enumerated type A user defined type created within a Java program by specifying its type name and allowable values within an `enum` statement

Event An asynchronous occurrence during a program's execution that can be used to redirect the execution path of a program

Event handler A method that is executed when an event occurs

Exception class The API class `Throwable` or a descendent of that class

Exception error message A string contained within an exception object that normally contains descriptive error information

Exception object An instance of the API class `Throwable`, or one of its decedents, which can be passed to a `catch` clause when an error is detected during the execution of a method

Exceptions A programming construct that promotes the reusability of methods by deferring the decision as to what action to take when an error condition is detected to the invoker of the method

Final class A class that cannot be a parent class; it cannot be extended

Flow chart A graphical representation of an algorithm

Fractal A mathematical or geometric object that has the property of self-similarity; that is, each part of the object is a smaller or reduced copy of itself

General solution Part of the methodology of formulating recursive algorithms; it is a solution to the original problem that uses the portion of the methodology known as the reduced problem

Generic class A class that contains generic methods and is coded in a way as to permit the type of its data members to be specified when an instance of the class is created

Generic method A method that can perform its algorithm on any type of object passed to it

Generic parameter A parameter that can be passed an object of any type and whose type is specified using a type placeholder

Generic parameter list A list of the type placeholders, coded within a method's signature, that are used in the method's parameter list

Generics A programming concept that promotes reusability by permitting the type of a method's parameters and returned value to be specified by the method's invoker and permitting the type of a class's data members to be specified when an instance of the class is created

Get method A method used to fetch the values of a class's private data members

Graphical User Interface (GUI) A means of interacting with the program user via a point-and-click mode, as opposed to a text-based mode, aimed at facilitating the I/O process

Hypertext Markup Language (HTML) A scripting language for writing instructions to be downloaded and executed by a Web browser to build and display a Web page; the script can contain instructions to download and execute a Java applet

Index The integer associated with a variable in an array

Inheritance A programming concept in which a new class can contain all of the data members and methods of an existing class by simply including an `extends` clause in its heading

Inner class A class that is defined within another class

Input method A method normally named `input` that ordinarily permits the program user to input the values of all of an object's data members

Instance of a class A specific object in the class

Integrated Development Environment (IDE) A program used by a programmer to develop a software product; it contains a collection of tools (e.g., a syntax checker, translator, editor, file-management system) that facilitate the development process

Interface A Java construct used to specify the signatures of related methods that are implicitly abstract and/or a declaration of public constants that are implicitly static and final

Iterator An object that can be used to perform time-efficient processing on all of the data elements contained in any data structure that imposes an ordering on its data elements

Java Development Kit (JDK) A set of tools used to develop Java programs; these tools include the API classes, a debugger, a compiler, an interpreter, an applet viewer, a documentation generator, a disassembler, various linking, loading, and binding tools, and a runtime environment

Java Virtual Machine A virtual computer system whose programming language is Java byte codes

Key A value associated with a data element that can be used to refer to the element

Layout manager A predefined protocol for the sizing and positioning of components added to a GUI container

Listener list An association of events and their event-handler methods that is part of a GUI component object

Local variable A variable defined within a code block whose scope is limited to the instructions within the code block

Loop A sequence of instruction that is repeated a specified number of times or until a Boolean value becomes true or false

Map A set of data structures that associate a key with each data element stored in the structure; the key can be used to specify the data element on which to operate

Menu mnemonic A menu shortcut key (hot key) associated with a terminal menu item

Methods The subprograms defined within a class, a sequence of instructions that perform a particular task

Multidimensional array An array in which each variable of the array is associated with 2, 3, … indices, for example an array of rows and columns

Multiple inheritance When a class inherits from more than one class; this is not supported in Java

Multitasking Executing several threads of an application at the same time or giving the impression that they are executing at the same time

Nested loops Coding loops inside of loops

Nested statements Statements that are contained within another statement or another statement's statement block

Non-void method A method that returns a value, whose type is specified in the method's signature

Object A particular occurrence of a class that contains all of the class's non-static data members

Object oriented programming (OOP) An approach to programming (a programming paradigm) aimed at facilitating the development of programs that deal with objects, such as starships, people, or Web pages

One-dimensional array An array in which each variable of the array is associated with one index

Operating system software A program to manage the resources of a computer system and to permit a user of the system to interact with it, usually via a point-and-click interface

Overloading methods The act of writing two or more methods in the same class that have the same name but different parameter lists

Overriding a method Rewriting an inherited method using the exact same signature of the inherited method

Parallel arrays A use of multiple one-dimensional arrays in which the ith element of each array is associated with the same entity; for example, if Mary's age was stored in the second element of one array, then the rest of Mary's information would be stored in the second element of the other arrays

Parameter A variable that can receive a value (an argument) passed to the method when it is invoked

Parameter list A sequence of parameter names, each proceeded by its type, separated by commas, and enclosed in a set of parentheses

Parent class A class that is inherited from, also known as a super or base class

Parsing The act of changing a string into a numeric; also the act of separating a string into its component parts that are separated by a specified delimiter

Platform A particular CPU model and operating system

Platform independence The concept that the programmer's translation of a program can be transmitted to, and then run on, any computer system

Polymorphism The ability of one invocation to morph itself into an invocation of a parent's version of a method or any of its children's versions of the method; rooted in the fact that a parent reference variable can refer to an instance of a child class

Pop-up menu A space-saving alternative to a menu-bar-based drop-down menu that remains invisible until the user performs a platform-dependent mouse or keyboard action on a GUI component

Precedence rules A specification of the order in which to perform a set of operations

Primitive variable A variable that can store a numeric value, a Boolean value, or one character; the type used in its declaration is one of the primitive types

Primitive type The Java types `byte`, `short int`, `long`, `float`, `double`, `char`, and `boolean`

Priority queue A queue that associates a priority with each of its data elements; the elements assigned the highest priority are fetched and deleted (on a first-in-first-out basis) before those of lower priority

Private data member A data member of a class that cannot be directly accessed by methods that are not part of the class; `get` and `set` methods are used to fetch and change their values

Producer A process that generates data

Pseudorandom numbers Apparent, but not truly, random numbers

Public data members Data members of a class that can be directly accessed by methods that are not part of the class; they are accessed by coding their name preceded by either the name of an instance of the class or the class name, followed by a dot

Queue A data structure in which the data elements are fetched and deleted on a first-in-first-out basis

Random access memory (RAM) High-speed, high-cost storage physically located in close proximity to the central processing unit

Recursion The act of defining something in terms of itself

Recursive method A method that invokes itself or initiates a sequence of method invocations that eventually leads to an invocation of itself

Reduced problem Part of the methodology of formulating recursive algorithms, it is a problem similar to the original problem, usually between the original problem and the base case, usually closer to the original problem, and (when progressively reduced) becomes the base case for all versions of the original problem

Reference variable A variable that can store a memory address; the type used in its declaration is the name of a class

Registering an event handler The act of associating an event-handler method with a particular event that could be performed on a GUI component

Runtime the time during which the program is in execution

Scope of a variable or a method The range of a program's instructions within which a variable can be used or a method can be invoked

Sentinel loop A loop that ends on a particular value of the data it is processing or on a particular user input; for example, a negative deposit

Serializing objects The act of disassembling objects before writing them to a disk file so they can be recreated when they are read from the disk

Set methods Methods used to change the values of a class's private data members

Shallow comparison Comparing the contents of one variable to the contents of another variable using the equality (==) operator

Shallow copy Copying the contents of one variable into another using the assignment (=) operator

Shared buffer Memory used to temporarily share a data item among one or more threads

Show method A method named `show` that ordinarily outputs all of the data members of an object or draws the object

Signature of a method The first line of a method's code

Software engineer A computer professional that produces programs that are error free, within budget, on schedule, and satisfy the customers' current and future needs

Stack A data structure in which the data elements are fetched and deleted on a last-in-first-out basis

States of a thread The six statuses a thread can assume from the time it is created to the time it is terminated

Static data member A class's data member that is designated to be shared by all instances of the class by including the keyword `static` in its declaration

Static method A method that is designated to be invoked by preceding the method name by the method's class name followed by a dot; they are intended to be methods that do not operate on instances of the class

String A finite sequence of characters

Subclass A class that inherits from (extends) another class, also known as a derived or child class

Super class A class that is inherited from, also known as a parent or base class

Swapping algorithm An algorithm that swaps the values contained in two variables

Synchronized buffer A buffer whose access is managed in a way that imposes protocols of proper access to the data on the threads that share the buffer

Syntax The rules for forming properly constructed program instructions; the grammar of a programming language

Text file A file whose information is intended to be characters and is therefore interpreted using the ASCII or Unicode tables; ordinarily the file extension `.txt` is appended to the file's name

Thread An independent execution path through a program

Token A component part of a string that is terminated by a specified delimiter, for example, a space

Tokenizing a string Extracting all of the tokens from a string

Top-level container The basic building block component of a graphical interface, which contains the other GUI components that make up the interface

`toString` method A method named `toString` whose task is to return the string representation of an object; ordinarily, the string contains the annotated values of all of an object's data members

Totaling or summation algorithm An algorithm that computes the sum of a set of numeric values by repeatedly adding each value to the subtotal of the values in the set that preceded it

Type placeholder Any valid identifier that is not the name of a class used within the application of which it is a part; a placeholder is used as a type of a generic parameter and can be used as a returned type

Unboxing A context-sensitive feature of Java in which an instance of a wrapper class object is replaced with the primitive value it contains

Unicode An expanded tabulation of characters and control characters and the bit patterns used to represent them

Universal modeling language (UML) diagram A graphical representation of a class that specifies the class's name, data members, and the signatures of its methods; it is used to design a class

Variable A named memory cell that can store a specific type of data item

Void method A method that does not return a value

Worker method A method that is invoked by another method to perform a specific task (work) for it; for example, fetching the value of one of an object's data members or drawing the object

Wrapper class An API class that contains non-static primitive data members of a particular type

Appendix G

Using the Online API Documentation

The documentation of the Application Programing Interface (API) is available online. To access it, you can Google: *Java API documentation* and click the link that begins with `docs.oracle.com/javase`, such as the one shown below:

http://docs.oracle.com/javase/7/docs/api/

To quickly locate the documentation on a particular class, you can Google the class's name and then click the link to the class's documentation. The following link was displayed after Googling *Java Math class*:

http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Clicking this link displays the information shown in Figure G.1, which is typical of the format of the documentation for any class. As shown in the figure, the class name is at the top of the documentation. Below it is the package that is imported into a class to gain access to the API class and its methods. This package name can be copied from the documentation and pasted into the class's file just before its class heading. It is preceded by the keyword **`import`** and followed by a semicolon.

Below the package name is the specification of the class's access and inheritance details. In the case of the `Math` class, this information indicates that the class's access is public, the class is final (which means it cannot be extended as a parent class), and its parent class is the class `Object`. Below that is a general description of the class.

Below the general descriptive information is a *Field Summary* (Figure G.2), which is a tabulation of the name and description of all of the data members contained in the class. This is followed by a *Method Summary*, which is a tabulation of the names of each method in the class and their parameter list followed a brief description of the method's functionality.

To the left of each data member's name in the Field Summary is its type, which may be preceded by the key word **`static`**. Static data members are accessed by preceding their name with the name of the class followed by a dot. To the left of each method's name in the Method Summary is the method's returned type, which may be preceded by the key word **`static`**. Static methods are invoked by preceding their name with the name of the class followed by a dot. Non-static methods are invoked by preceding their name with the name of an instance of the class followed by a dot.

More detained documentation on a data member or a method can be displayed by clicking the name of the data member in the Field Summary or the name of the method in the Method Summary. Figure G.3 was displayed when the method name `acos`, shown at the bottom of Figure G.2, was clicked.

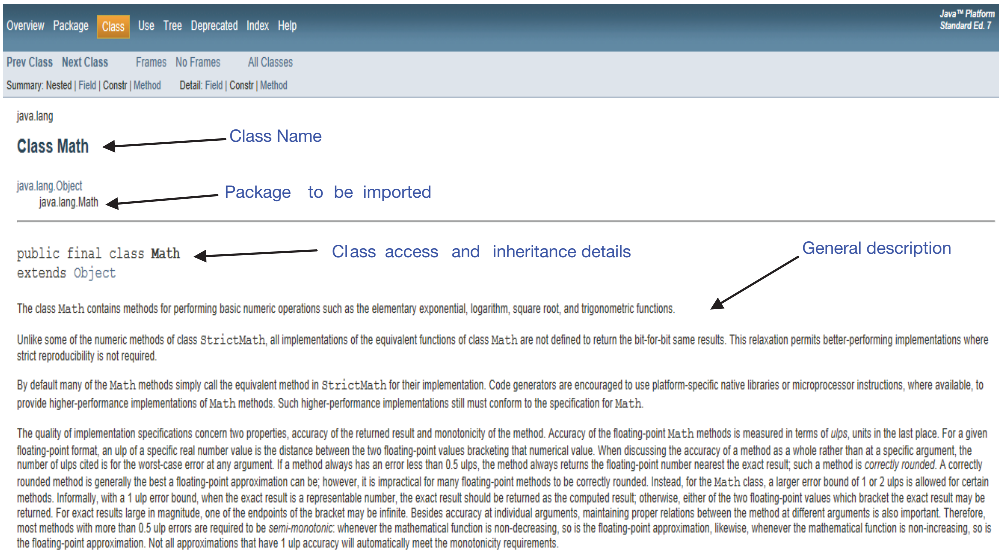


Figure G.1
The top portion of the online documentation of the **Math** class.

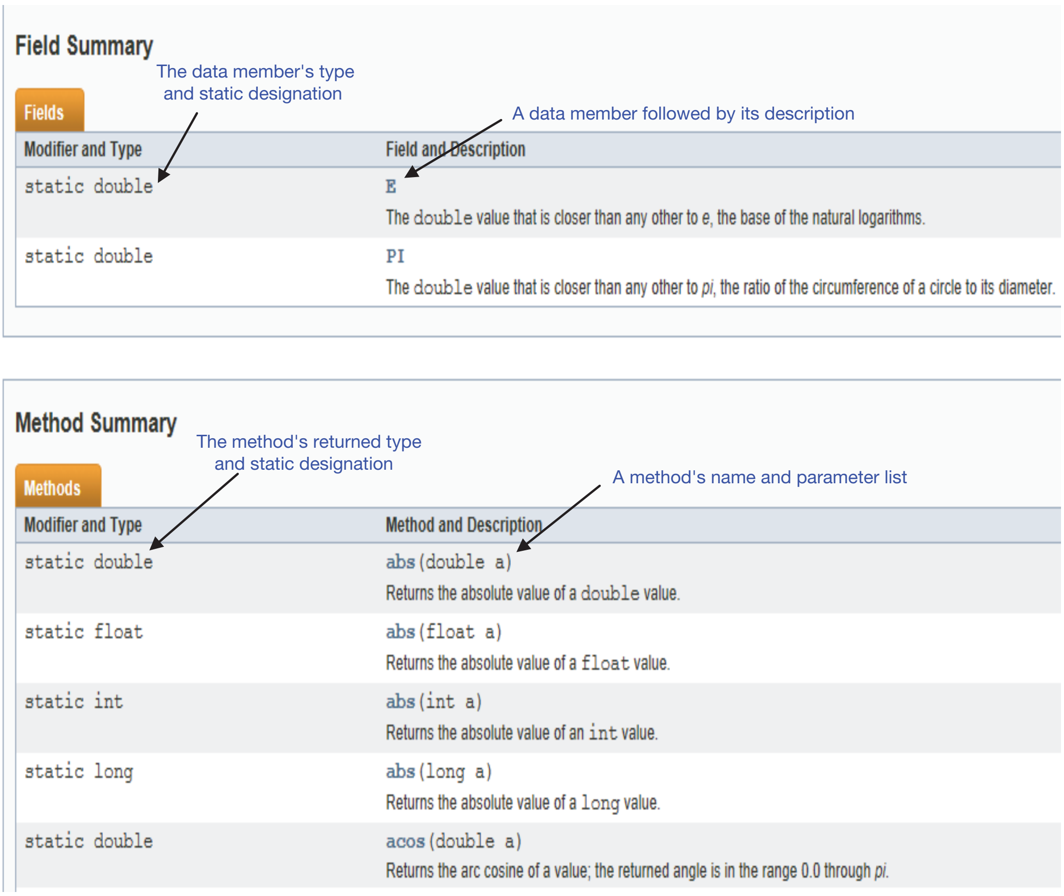


Figure G.2
The partial Field Summary and Method Summary of the API Math class.

acos

```
public static double acos(double a)
```

Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*. Special case:

- If the argument is NaN or its absolute value is greater than 1, then the result is NaN.

The computed result must be within 1 ulp of the exact result. Results must be semi-monotonic.

Parameters:

`a` - the value whose arc cosine is to be returned.

Returns:

the arc cosine of the argument.

Figure G.3
Detailed documentation of the Math class's **`acos`** method.

APPENDIX H

SOLUTIONS TO SELECTED ODD KNOWLEDGE EXERCISES

CHAPTER 1

1. **(b)** The number of computers grew from 200 to 800 – a factor of 4
3. Operating systems (such as: Windows. Linux or Apple OS X) are the instructions used by the computer system to schedule tasks, to allocate memory and other system resources, to detect errors and to perform other computer system functions. Application software is commonly used by the human user, while system software is used by the computer system. Examples of application software include word processors, spreadsheets, mail readers, Web browsers, and game programs.
5. Both a and c are characteristics of secondary storage which is nonvolatile, has a very large capacity and is slower and cheaper than RAM.
7. **(a)** A device that is only used for output is a printer or a speaker.

   (c) Devices used for both input and output include touch screens, flash drives, floppy disks, and writable CDs, DVDs.
9. The computer as we know it today was the work of many people over hundreds of years, beginning with the development of early calculating machines: the abacus, the slide rule, Napier's bones and the Pascaline. The modern computer was based on the designs of Babbage, von Neumann, Mauchly, and Eckert. Lady Ada Lovelace and Grace Hopper were pioneers in the field of programming languages. Metcalfe and Boggs, Cerf and Kahn and Berners-Lee connected computers together into networks, the Internet and the World Wide Web, respectively.
11. **(b)** loses its contents if power is interrupted.
13. **(c)** chips, replacing the larger transistor circuits.
15. **(a)** First programmer - Lady Ada Augusta Byron, the Countess of Lovelace

    (b) Inventor of the Java programming language – James Gosling
17. Platform independence is the ability of software to run on any computer system or platform. Every manufacturer's chipset has its own unique machine language and therefore usually requires its own translating program to translate from source code instructions to its machine language. Java achieves platform independence by compiling the source code instructions into byte code, which is later translated on the end user's computer into its own specific machine code.
19. A class is a group of related data members and member methods. It is the template used to create an object. An object is a particular instance of a class. From one class we can create an unlimited number of objects or instances of the class, just as with a blueprint we can create many houses, or with a cookie cutter, we can create many batches of cookies.
21. **(a)** CPU – central processing unit **(c)** I/O – input/output

    (e) JVM – Java virtual machine **(g)** GUI- graphical user interface
23. **(d)** (5, 30) since it is 5 pixels to the left of the left boundary and 30 pixels below the top.
25. **(1)** character data, **(2)** translated instructions, and **(3)** numeric data.
27. **(a)** 01010011 = 83 in decimal **(b)** 00101111 = 47 in decimal

CHAPTER 2

1. **(a)** False, the contents of the variable may change but the data type does not

   (c) False **(e)** False
3. A variable is a named memory cell that stores one data item that can change during program execution. Primitive variables can store a single numeric data value, one character, or one Boolean truth value. Reference variables store (RAM) memory addresses.

5. **(a)** `boolean false` **(c)** `double` 0.0

7. Numeric literals, containing decimals such as 19.5, are assumed to be type double. If a numeric literal is to be assigned to a `float` variable, the letter `f` for float, must be appended to the literal to inform the translator that a loss of precision is acceptable, otherwise an error results. (This is a correct declaration `float weight = 19.5f;`)

9. **(a)**
```
System.out.println("Sara Larson");
System.out.println("Smalltown, USA");
```
(b)
```
System.out.println("Sara Larson \nSmalltown, USA \n");
```
(a)
```
System.out.printf("Sara Larson \n");
System.out.printf("Smalltown, USA \n");
```
(b)
```
System.out.printf("Sara Larson \nSmalltown, USA \n\n");
```

11.

The `String` object at memory address 1024

```
int distance = 675; String myName = "Jane";
```

13. **(a)** 17 - 5 * 2 + 12 = 19 **(c)** (48 + 12) / 12 + 18 * 2 = 41
(d) 21 - 9 + 18 + 4 * 3.7 = 44.8

15.
```
double average = ((double)(55 + 57 + 60)) / 3;
```

17. **(a)** True **(b)** False, it is used for output
(c) True **(d)** False, it would return the empty string ("")
(e) True

19.
```
sBalance = JOptionPane.showInputDialog("Type your current " + "checking account
                                        balance");
```

21.
```
double  deposit;
deposit = Double.parseDouble(sDeposit);
```

CHAPTER 3

1. **(a)** True **(b)** False, it is the method signature
(c) False **(d)** False, this method returns a value

3. **(a)** The signature of a method that does not operate on an object must contain the keyword `static`.
(c) When we invoke a static method, we begin the invocation statement with the name of the class followed by a dot.

5. **(a)** True
(c) False, the client method sends an argument into the worker method's parameter
(e) False, a method can only return a single value
(g) False
(h) False, value parameters

7. The statement following the statement that invoked the method executes next.

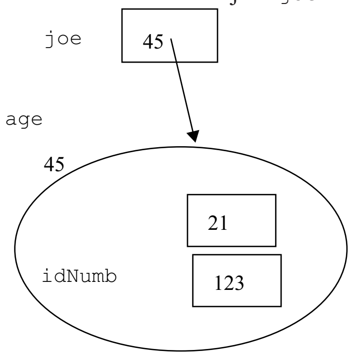
9. `static double checkAmount;`

11. **(a)** `drawRect` **(c)** `drawOval`
(e) `fillOval`, using the same value for the height and width

13. **(a)** House is to object as blueprint is to class.

(c) The name of the graphic used to specify a class is a UML diagram.
(e) Member methods of a class are usually designated to have private access.

15. (a) The address of the object `joe`
(c)

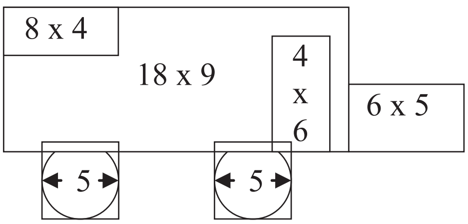



```
Person joe = new Person();
```

17. (a)
```
public CoffeeCup(int size, double price)
{   this.size = size;
    this.price = price;
}
```
(b) `CoffeeCup cup1 = new CoffeeCup(8, 3.85);`
(c) `System.out.println(cup1);`
(d) `System.out.println(cup1.tostring();`
(e) CoffeeCup@456af2 (the address, 456af2, will probably be different)
(f) `g.drawString(cup1.toString(), 200, 250);`

19.
```
public void show(Graphics g)
{
    g.drawString("size is: " + size, 250, 250);
    g.drawString("price is: " + price, 250, 280);
}
```

20.



21. This might be a typical response based on the model given in Exercise 20

Component	Shape	Shape's X or Line's X_1 coordinate	Shape's Y Line's Y_1 coordinate	Width Line's X_2 coordinate	Height Line's Y_2 coordinate
window	rectangle	x	y	8	4
body	rectangle	x	y	18	9
door	rectangle	x + 14	y + 2	4	6
cab	rectangle	x + 18	y + 4	6	5
rear tire	circle	x + 3	y + 8	5	5
front tire	circle	x + 11	y + 8	5	5

23. (a) `this.total = total * 2;`
(c) `public void setTotal(int total)`

```
{
  this.total = total;
}
```

(e) `int currentTotal = myAccount.getTotal();`
`myAccount.setTotal(currentTotal * 2);`

(g)
```
public void toString()
{
  System.out.println("The total is: " + total );
}
```

(i) `private`

25. **(a)** `static Starship largest(Starship ship1, Starship ship2);`
(b) `ship1 = largest(ship1, ship2);`
(c) The new color.
(d) `public boolean sameModel(Starship ship1, Starship ship2);`
(e) `isSame = sameModel(ship1, ship2);`

CHAPTER 4

1. **(a)** True **(c)** False **(e)** True

3. Method invocations and control-of-flow (or control) statements, such as decision and loops, alter the execution path.

5.
```
if (myBalance ==10.0)
{
   System.out.println(myBalance);
}
 else
{
   System.out.println("my balance is not 10.0");
}
```

7. **(a)** True
(c) True, although it can be empty
(e) True
(g) True

9. **(a)** False, but it is good programming style to include a default statement
(c) True
(e) True
(g) False, it can only be written as a switch statement if the selection statements are of the appropriate type

11.
```
if(item.equals("Hamburger"))
{
   System.out.println("You ordered a Hamburger.");
}
else if(item.equals("Taco"))
{
   System.out.println("You ordered a Taco.");
}
else if(item.equals("BLT"))
{
   System.out.println("You ordered a BLT sandwich.");
}
else
```

```
{
   System.out.println("You did not place a valid order.");
}
```

13. **(a)**
```
Scanner consoleIn = new Scanner(System.in);
System.out.println("Type the year of your birth: ");
int birthYear;
birthYear = consoleIn.nextInt();
```
(b)
```
Scanner consoleIn = new Scanner(System.in);
String name;
System.out.print("Enter your name: ");
name = consoleIn.nextLine();
```

15. **(a)** True **(b)** False **(c)** True

17. **(a)**
```
File fileObject = new File("e:/Dates.txt");
Scanner fileIn = new Scanner(fileObject);
int year;
year = fileIn.nextInt();
```
(b)
```
File fileObject = new File("e:/Names.txt");
Scanner fileIn = new Scanner(fileObject);
String name;
int age;
age = fileIn.nextInt();
fileIn.nextLine();  // to flush the buffer
name = fileIn.nextLine();
```

19.
```
import java.io.*;
public class DiskIO
{
  public static void main(String[] args) throws IOException
  {
    double myBalance =2567.00;
    double yourBalance = 3876.25;
    new // if file exists it will be deleted
    FileWriter fileWriterObject = new FileWriter("c:/Balances.txt");
    PrintWriter fileOut = new PrintWriter(fileWriterObject, false);

    fileOut.println(myBalance + " " + yourBalance);
    fileOut.close();
  }
}
```

21.
```
inputfile.close();
```

CHAPTER 5

1. **(a)** False, it is possible for a while loop body not to execute at all
(c) True
(e) False, it is a pretest loop
(g) True
(i) True
(k) False, when the Boolean condition becomes false.
(m) True, since the loop is never entered the loop control variable is not changed
(o) True

3.
```
int n;
int count =1;
```

```
int sum =0;
String instring;
instring = JOptionPane.showInputDialog("Type a number: " );
  n = Integer.ParseInt(instr);
  while(count <= n)
  {
    if (count % 2 ==0)     //number is even
    {
        sum = sum + count;
    }
    count++;
  }  //end while
  JoptionPane.showMessageDialog(null, "The sum of even integers " +
                               "from 1 to " + n + " is " + sum);
```

5. (a) The value of `i` is never equal to 20, so the loop never terminates.
   (b) Because the loop does not terminate the output statement after the loop is never reached and is not executed.

7. (a) Output: 8, 5, 2, -1
   (b)
```
for (int x = 8; x >= -1; x = x - 3)
{
 System.out.println(x);
}
```

9.
```
int trys = 0;
int input;
String sInput;
do
{
   trys++;
   sInput = JOptionPane.showInputDialog("Enter a number from 0 to 5");
   input = Integer.parseInt(sInput);
   if(input >= 0 && input <= 5)
   {
      JOptionPane.showInputDialog("Thanks for the valid input");
      break;
   }
   else
   {
      JOptionPane.showInputDialog("invalid input");
   }
} while(trys < 3);
```

11. (a)
```
int randomNumber;
Random randomObject1 = new Random(); // uses time of day
for(int i=1; i<=20; i++)
{
    randomNumber = randomObject1.nextInt(501);
    System.out.print(randomNumber + " ");
}

SecureRandom randomObject2 = new SecureRandom();
System.out.println();
for(int i=1; i<=20; i++)
{
    randomNumber = randomObject2.nextInt(501);
    System.out.print(randomNumber + " ");
}
```

(c)
```
int randomNumber;
int min =7;
int max =500;
Random randomObject = new Random(2468); // uses seed
for(int i=1; i<=20;, i++)
{
   randomNumber = min + randomObject.nextInt(500 - min + 1);
   System.out.print(randomNumber);
}
```

CHAPTER 6

1. **(a)** True **(c)** True
(e) True **(g)** False, arrays can be multi-dimensional
(i) True

3. An array element is a reference variable, while a non-array element may be a primitive or a reference variable. An array variable is able to store many elements, while a primitive variable only stores one. An array variable uses square brackets ([ ]) and an index to indicate the position of an element in the array, while a non-array variable does not.

5. **(a)** True **(c)** False, `gameScores[99]`
(e) False, 100 **(g)** `System.out.println(gameScores[99]);`
(i)
```
int total = 0;
for(int i = 0; i <  gameScores.length; i++)
{
    total = total + gameScores[i];
}
System.out.println(total / gameScores.length);
```

7. **(a)** 45 **(c)** 4
(e) y[4] = y[4] + 20.5; **(f)** z = y[0] + y[1] +y[2];

9. **(a)**
```
String[] names = new String[50];
double[] weights = new double[50];
double[] targetWeights = new double[50];
```

(c)
```
for(int i = 0; i < names.length; i++)
{
   if(names[i].equalsIgnoreCase("joe smith")
   {
      System.out.println(weight[i] + " " + targetWeight[i]);
   }
}
```

CHAPTER 7

1. **(a)** True **(c)** True
(e) False, a deep copy **(g)** True

3. A shallow comparison compares reference variables or the addresses of two objects to determine if they refer to the same object or two different objects. A deep comparison compares the contents of the data members of two objects to determine if they are the same.

5. Explain the difference between a deep copy and a clone. A deep copy copies the values of the data members of one object into the data members of another object, using the `set` method. When an object is cloned, a new instance of the object's class is created, and the values of all of an existing object's data members are copied into the corresponding data members of the new object. There are now two objects instead of one.

7.
```
if(s1 != s2)
{
   System.out.println(" Two objects");
}
```

9. (a) *6* (c) *Hello everyone*
   (e) *Hello*

11. Aggregation is combining objects so that the instance of one class is a field in another class. It establishes a "has a" relationship.

13. Use the API BigInteger class to create a BigInteger object.
```
BigInteger num1 =  new BigInteger ("12345678910111213345678 9");
```

15. Invoke the BigInteger multiply method on the BigInteger object. For example,
```
BigInteger num1 =  new BigInteger ("123456789101112133456789");
BigInteger num2 =  new BigInteger.valueOf(2);
BigInteger largenum = num1.multiply(num2);
```

17. (a) 2
   (b) `CarColor favoriteColor = CarColor.BLUE;`
   (c) `System.out.println(CarColor.BLUE + " " + favoriteColor.ordinal());`

CHAPTER 8

1. (a) False (c) True
   (e) False, constructors are not inherited (g) True
   (i) False, they are overloaded (k) True
   (m) False, all we need are the class's byte codes

3. Reduced coding time: a parent class can collect functionality and data members common to several classes into one class so they need only be coded once in the parent class.

   Code reusability: A child can inherit all of the data members and methods of a previously developed class not coded as part of its program, and then add methods and data members or overwrite methods that are not suited for its applications.

5. Public: Child classes and client code have direct access. Protected: Child classes have direct access but not client code in a separate package. Private: Neither child classes nor client code have direct access.

7. `super.input();`

9. Declare the method to be final.

11. When you wanted to expand its parameter list.

13. An abstract class is used to collect all of the data members and methods that are common to two or more classes that will make up a program. The classes simply extend it, and then add the data members and methods specific to them to it.

15. `Transporter[] vehicles = new Transporter[200];`

17. An interface can contain the signatures of related methods that are implicitly abstract and/or declarations of public constants that are implicitly static and final. An advantage of an interface is that any class that implements the interface must implement all of the methods defined in the interface.

19. Include the `implements ManyMethods` clause in the adapter class's signature, and implement all 20 of the methods defined in the interface with empty code blocks.

CHAPTER 9

1. **(a)** True
   (c) False, usually the most difficult part is the discovery of the reduced problem
   (e) True, if the base case is not realized
   (g) False; typically they are slower than their loop base (iterative) counterparts because of the time required to transfer execution to the recursive invocations they make
3. The symbol with the number 5 to its left
5. Iterative: f1 = 1; f2 = 1; f3 = 1 + 1 = 2; f4 = 1 + 2 = 3; f5 = 2 + 3 = 5; f6 = 3 + 5 = 8;
   f7 = 5 + 8 = 13; f8 = 8 + 13 = 21;
   Non-iterative: f8 = f7 + f6 = (f6 + f5) + (f5 + f4) =
   (f5 + f4) + (f4 + f3) + (f4 + f3) + (f3 + 1) =
   (f4 + f3) + (f3 + 1) + (f3 + 1) + (1 + 1) + (f3 + 1) + (1 + 1) (1 + 1) + 1 =
   (f3 + 1) + (1 + 1) + (1 + 1) + 1 + (1 + 1) + 1 + (1 + 1) + (1 + 1) + 1 +
   (1 + 1) (1 + 1) + 1 =
   (1 + 1) + 1 + (1 + 1) + (1 + 1) + 1 + (1 + 1) + 1 + (1 + 1) + (1 + 1) + 1 +
   (1 + 1) (1 + 1) + 1 =
   21
7. Dynamic programming
9. Base case: `if(m == n) return n;`
   Reduced problem: sum of the even integers from m-2 to n
   General Solution: m + the reduced problem
11. Because that is the base case, which halts the recursive invocations.
13. Combine the base case, reduced problem and general solution into a recursive algorithm, using a flow chart similar to the one shown in Figure 9.6
15. To move six rings: $2^6 - 1$ For ten rings: $2^{10} - 1$ For n rings: $2^n - 1$

CHAPTER 10

1. **(a)** True **(c)** True
   (e) True **(h)** False
   (j) False, but if the exception is a checked exception the method's signature must contain a `throws` clause
   (l) True **(n)** False
3. When the error that caused the problem is a serious error, because the translator will then warn the programmer that a catch block to deal with the problem was not included in the program that invoked the method.
5. `Exception`
7. **(a)** Checked **(c)** Checked
   (e) Unchecked **(g)** Unchecked
9. Invoke the `getMessage` method on the exception object passed to the catch clause:

```
String error = e.getMessage();
```

CHAPTER 11

1. **(a)** False, it stands for Graphical User Interface
   (c) True
   (e) True
   (g) True
   (i) False, a `Pane` container should be used
   (k) True

3. Buttons are used to initiate processing; Radio buttons are used to select one item from a set of mutually exclusive items; Check boxes are used select to one or more items from a set of items.
5. North, west, center, east, and south
7. `Pane root = new Pane();`
   `root.setStyle("-fx-background-color: red");`
   `Scene scene = new Scene(root, 600, 650);`
9. `compute.setOnAction(e -> clickHandler(e));`
11. `scene.setOnMouseClicked(e -> clickHandler(e));`
13. `scene.setOnKeyreleased(e -> keyHandler(e));`

CHAPTER 12

1. (a) False, they are normally used to select one input from a set of mutually exclusive inputs
   (c) False, only one selection can be made
   (e) True
   (g) True
   (i) False, they are defined in an `ObservableList`
   (k) True
   (m) False
3. A combo box is used to select one item from a set of items; lists are used to select one or more items from a set of items.
5. Only one item can be selected from a combo box, one or more values can be selected from a list. The items in a combo box are displayed when the arrow in its drop-down button is clicked. A list is displayed with a scroll bar by default when the size of the list box is too small to display all of its values.
7. Menus take up very little space in a window, because their items are only displayed when the user indicates that she wants to use them by clicking them.
9. The path to the selected file.
11. `salad.setOnAction(e -> saladClickHandler());`
13. `if(cb1.isSelected() == true)`

CHAPTER 13

1. (a) True
   (c) False
   (e) False
   (g) True
   (i) True
   (k) True
3. `T1[] copy;`
   `copy = Arrays.copyOf(values, values.length);`
5. (a) True
   (c) True
   (f) True
7. `public class G7Class <T> implements Comparable<GClass>`
9. `ArrayList <String> s2;`
11. `HashMap`, or `TreeMap`, or `LinkedHashMap`
13. `PriorityQueue <PhoneListing> pl = new PriorityQueue <PhoneListing>();`

CHAPTER 14

1. (a) True, until the program it is part of ends
   (c) True
   (e) True, or they can extend the class `Thread` (which implements the interface `Runnable`).
   (g) False

3. New, runnable, waiting, timed waiting, blocked, and terminated
5. The wait method
7. The method invokes the `wait` method, or the method invokes the `sleep` method.
9. **(a)** True **(c)** False
   (e) False **(g)** False
11. The consumer task is using data generated by another task, the producer task. Two problems can occur. The producer generates a data item and overwrites a previously generated data item not yet processed by the consumer task, or the consumer reprocess a previously processed data item (or a data item containing a default value) because the producer has not generated a new data item.
13. Use synchronized methods or synchronized statements.

Index

O

P

Q

R