

Cay Horstmann
Java Concepts
3/e Late Objects

Includes Java 8 coverage

WILEY

Variable and Constant Declarations

```
Type      Name      Initial value  
 /       /       /  
int cansPerPack = 6;  
  
final double CAN_VOLUME = 0.335;
```

Method Declaration

Modifiers	Return type	Parameter type and name
/ \ /	/	/ /

```
public static double cubeVolume(double sideLength)  
{  
    double volume = sideLength * sideLength * sideLength;  
    return volume;  
}  
} Exits method and returns result.
```

Mathematical Operations

Math.pow(x, y)	Raising to a power x^y
Math.sqrt(x)	Square root \sqrt{x}
Math.log10(x)	Decimal log $\log_{10}(x)$
Math.abs(x)	Absolute value $ x $
Math.sin(x)	Sine, cosine, tangent of x (x in radians)
Math.cos(x)	
Math.tan(x)	

Selected Operators and Their Precedence

(See Appendix B for the complete list.)

[]	Array element access
++ -- !	Increment, decrement, Boolean not
* / %	Multiplication, division, remainder
+ -	Addition, subtraction
< <= > >=	Comparisons
== !=	Equal, not equal
&&	Boolean and
	Boolean or
=	Assignment

Loop Statements

Condition
/

```
while (balance < TARGET)  
{  
    year++;  
    balance = balance * (1 + rate / 100);  
} Executed while condition is true
```

Initialization	Condition	Update
/	/	/

```
for (int i = 0; i < 10; i++)  
{  
    System.out.println(i);  
}
```

String Operations

```
String s = "Hello";  
int n = s.length(); // 5  
char ch = s.charAt(1); // 'e'  
String t = s.substring(1, 4); // "ell"  
String u = s.toUpperCase(); // "HELLO"  
if (u.equals("HELLO")) ... // Use equals, not ==  
for (int i = 0; i < s.length(); i++)  
{  
    char ch = s.charAt(i);  
    Process ch  
}
```

Conditional Statement

```
if (floor >= 13) Condition  
{  
    actualFloor = floor - 1; } Executed when condition is true  
}  
else if (floor >= 0) Second condition (optional)  
{  
    actualFloor = floor;  
}  
else  
{  
    System.out.println("Floor negative"); } Executed when all conditions are false (optional)  
}
```

Class Declaration

```
public class BankAccount  
{  
    private double balance; } Instance variables  
    private int transactions;  
  
    public BankAccount(double initialBalance) Constructor  
{  
        balance = initialBalance;  
        transactions = 1;  
    }  
  
    public void deposit(double amount) Method  
{  
        balance = balance + amount;  
        transactions++;  
    }  
    . . .
```

do { *Loop body executed at least once*
System.out.print("Enter a positive integer: ");
input = in.nextInt();
} while (input <= 0);

Set to a new element in each iteration

An array or collection
for (double value : values) *Executed for each element*
{
 sum = sum + value;
}

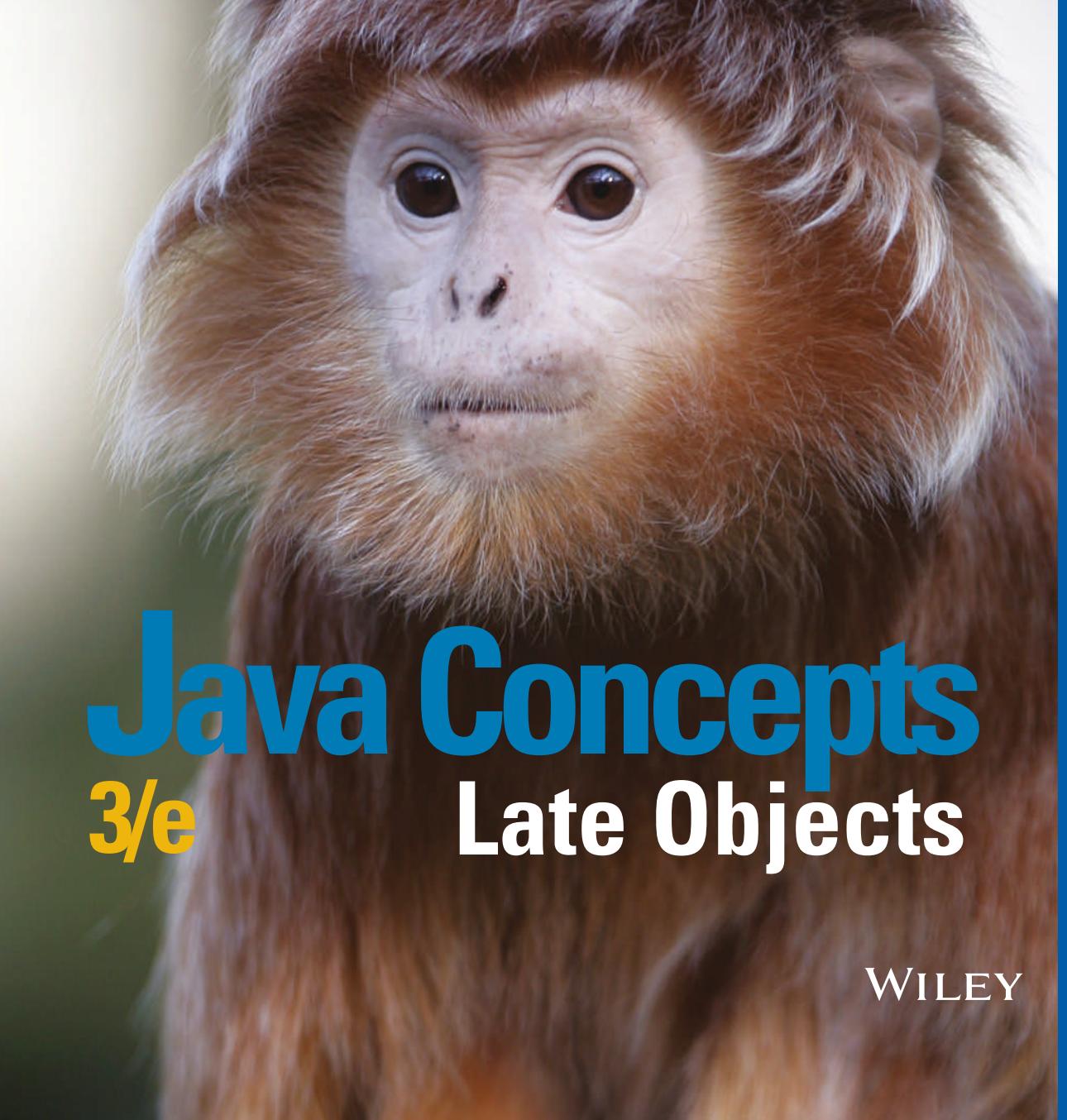
Java Concepts

Late Objects

3/e

Cay Horstmann

San Jose State University



Java Concepts

3/e Late Objects

WILEY

PUBLISHER	Laurie Rosatone
EDITORIAL DIRECTOR	Don Fowley
DEVELOPMENTAL EDITOR	Cindy Johnson
ASSISTANT DEVELOPMENT EDITOR	Ryann Dannelly
EXECUTIVE MARKETING MANAGER	Dan Sayre
SENIOR PRODUCTION EDITOR	Laura Abrams
SENIOR CONTENT MANAGER	Valerie Zaborski
EDITORIAL ASSISTANT	Anna Pham
SENIOR DESIGNER	Tom Nery
SENIOR PHOTO EDITOR	Billy Ray
PRODUCTION MANAGEMENT	Cindy Johnson
COVER IMAGES	(tiger) © Aprison Photography/Getty Images, Inc.; (bird) © Nengloveyou/Shutterstock; (monkey) © Ehlers/iStockphoto; (rhino) © irawansubingarphotography/Getty Images, Inc.

This book was set in Stempel Garamond LT Std by Publishing Services, and printed and bound by Quad/Graphics, Versailles. The cover was printed by Quad/Graphics, Versailles.

This book is printed on acid-free paper. ∞

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

Copyright © 2017, 2013, 2010 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201) 748-6011, fax (201) 748-6008, website <http://www.wiley.com/go/permissions>.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return shipping label are available at www.wiley.com/go/returnlabel. If you have chosen to adopt this textbook for use in your course, please accept this book as your complimentary desk copy. Outside of the United States, please contact your local representative.

ISBN 13: 978-1-119-32102-6

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

PREFACE

This book is an introduction to Java and computer programming that focuses on the essentials—and on effective learning. The book is designed to serve a wide range of student interests and abilities and is suitable for a first course in programming for computer scientists, engineers, and students in other disciplines. No prior programming experience is required, and only a modest amount of high school algebra is needed. Here are the key features of this book:

Present fundamentals first.

The book takes a traditional route, first stressing control structures, methods, procedural decomposition, and arrays. Objects are used when appropriate in the early chapters. Students start designing and implementing their own classes in Chapter 8.

Guidance and worked examples help students succeed.

Beginning programmers often ask “How do I start? Now what do I do?” Of course, an activity as complex as programming cannot be reduced to cookbook-style instructions. However, step-by-step guidance is immensely helpful for building confidence and providing an outline for the task at hand. “Problem Solving” sections stress the importance of design and planning. “How To” guides help students with common programming tasks. Additional Worked Examples and Video Examples are available online.

Problem solving strategies are made explicit.

Practical, step-by-step illustrations of techniques help students devise and evaluate solutions to programming problems. Introduced where they are most relevant, these strategies address barriers to success for many students. Strategies included are:

- Algorithm Design (with pseudocode)
- Tracing Objects
- First Do It By Hand (doing sample calculations by hand)
- Flowcharts
- Selecting Test Cases
- Hand-Tracing
- Storyboards
- Solve a Simpler Problem First
- Adapting Algorithms
- Discovering Algorithms by Manipulating Physical Objects
- Patterns for Object Data
- Estimating the Running Time of an Algorithm

Practice makes perfect.

Of course, programming students need to be able to implement nontrivial programs, but they first need to have the confidence that they can succeed. This book contains

a substantial number of self-check questions at the end of each section. “Practice It” pointers suggest exercises to try after each section. And additional practice opportunities, including code completion questions and skill-oriented multiple-choice questions, are available online.

A visual approach motivates the reader and eases navigation.

Photographs present visual analogies that explain the nature and behavior of computer concepts. Step-by-step figures illustrate complex program operations. Syntax boxes and example tables present a variety of typical and special cases in a compact format. It is easy to get the “lay of the land” by browsing the visuals, before focusing on the textual material.



© Terraxplorer/iStockphoto.

Focus on the essentials while being technically accurate.

An encyclopedic coverage is not helpful for a beginning programmer, but neither is the opposite—reducing the material to a list of simplistic bullet points. In this book, the essentials are presented in digestible chunks, with separate notes that go deeper into good practices or language features when the reader is ready for the additional information. You will not find artificial over-simplifications that give an illusion of knowledge.

Visual features help the reader with navigation.

Reinforce sound engineering practices.

A multitude of useful tips on software quality and common errors encourage the development of good programming habits. The focus is on test-driven development, encouraging students to test their programs systematically.

Engage with optional science and business exercises.

End-of-chapter exercises are enhanced with problems from scientific and business domains. Designed to engage students, the exercises illustrate the value of programming in applied fields.

New to This Edition

Updated for Java 8

Java 8 introduces many exciting features, and this edition has been updated to take advantage of them. Interfaces can now have default and static methods, and lambda expressions make it easy to provide instances of interfaces with a single method. The sections on interfaces and sorting have been updated to make these innovations optionally available.

In addition, Java 7 features such as the try-with-resources statement are now integrated into the text.

Optional JavaFX Coverage

For those who prefer to use JavaFX instead of Swing, there is a new online resource that covers graphical user-interface programming with JavaFX.

Interactive Learning

Additional interactive content is available that integrates with this text and immerses students in activities designed to foster in-depth learning. Students don't just watch animations and code traces, they work on generating them. The activities provide instant feedback to show students what they did right and where they need to study more. To find out more about how to make this content available in your course, visit <http://wiley.com/go/bjlo2interactivities>.

2. Consider the following code segment:

```
if (hour < 21)
{
    response = "Goodbye";
}
else
{
    response = "Goodnight";
}
```

Determine the value of `response` when `hour` has the values given in the table below.

hour	response
20	
22	
21	
3	

Complete the second column. Press Enter to submit each entry.

6. Assume that weekdays are coded as 0 = Monday, 1 = Tuesday, ..., 4 = Friday, 5 = Saturday, 6 = Sunday. Rearrange the lines of code so that `weekday` is set to the next working day (Monday through Friday). Not all lines are useful.

Order the statements by dragging them into the left window. Use the guidelines for proper indenting and use of braces.

Statement	Order
if (weekday < 4)	1
{	2
weekday++;	3
}	4
else	5
{	6
.....	7

InterActivities

1. In this activity, observe the inputs. They denote hours in "military time" between 0 and 23. For each input, click on the appropriate line inside the `if` statement.

Please click on the next line.

```
int hour = in.nextInt();
if (hour < 12)
{
    greeting = "Good morning";
}
else
{
    greeting = "Good afternoon";
}
```

hour	greeting
4	Good morning
13	

0 errors

Start over

“CodeCheck” is an innovative online service that students can use to work on programming problems. You can assign exercises that have already been prepared, and you can easily add your own. Visit <http://codecheck.it> to learn more and to try it out.

A Tour of the Book

This book is intended for a two-semester introduction to programming that may also include algorithms, data structures, and/or applications.

Part A: Fundamentals (Chapters 1–7)

The first seven chapters follow a traditional approach to basic programming concepts. Students learn about control structures, stepwise refinement, and arrays. Objects are used only for input/output and string processing. Input/output is covered in

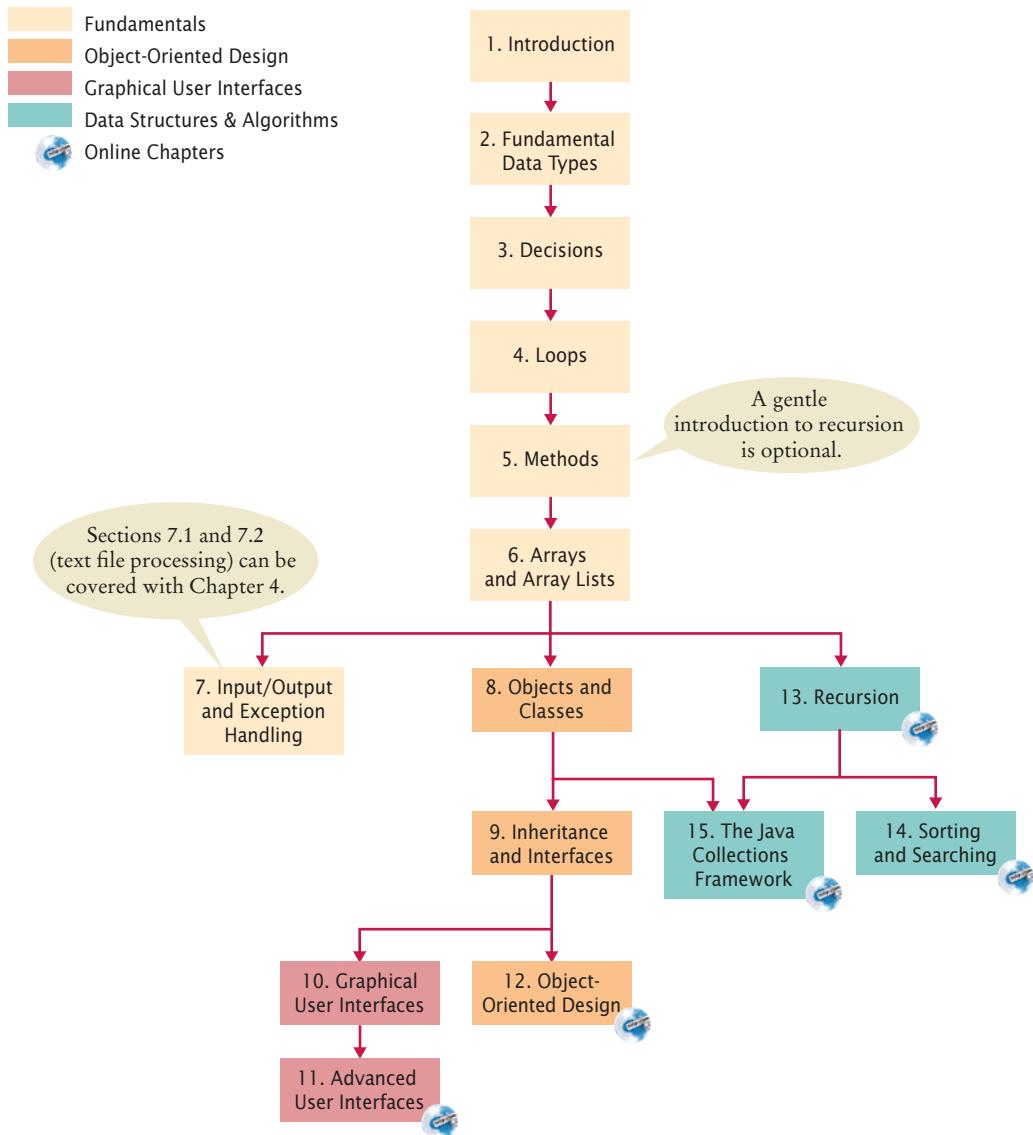


Figure 1 Chapter Dependencies

Chapter 7, but Sections 7.1 and 7.2 can be covered with Chapter 4; in that way, students can practice writing loops that process text files. Chapter 4 also provides an optional introduction to programming drawings that consist of lines, rectangles, and ovals, with an emphasis on reinforcing loops.

Part B: Object-Oriented Design and Graphics (Chapters 8–12)

After students have gained a solid foundation, they are ready to tackle the implementation of classes in Chapter 8. Chapter 9 covers inheritance and interfaces. A simple methodology for object-oriented design is presented in Chapter 12. Object-oriented design may also be covered immediately after Chapter 9 by omitting the GUI versions of the sample programs. By the end of these chapters, students will be able to implement programs with multiple interacting classes.

Graphical user interfaces are presented in Chapters 10 and 11. The first of these chapters enables students to write programs with buttons, text components, and simple drawings. If you want to go deeper, you will find layout management and additional user-interface components in the second chapter. Online versions of these chapters cover JavaFX instead of Swing.

Part C: Data Structures and Algorithms (Chapters 13–15)

Chapters 13–15 cover algorithms and data structures at a level suitable for beginning students. Recursion, in Chapter 13, starts with simple examples and progresses to meaningful applications that would be difficult to implement iteratively. Chapter 14 covers quadratic sorting algorithms as well as merge sort, with an informal introduction to big-Oh notation. In Chapter 15, the Java Collections Framework is presented from the perspective of a library user, without revealing the implementations of lists and maps. You can cover this chapter anytime after Chapter 8. Chapters 11–15 are available in electronic form on the Web.

Any subset of these chapters can be incorporated into a custom print version of this text; ask your Wiley sales representative for details.

Appendices

Many instructors find it highly beneficial to require a consistent style for all assignments. If the style guide in Appendix E conflicts with instructor sentiment or local customs, however, it is available in electronic form so that it can be modified. Appendices E–J are available on the Web.

- A. The Basic Latin and Latin-1 Subsets of Unicode
- B. Java Operator Summary
- C. Java Reserved Word Summary
- D. The Java Library
- E. Java Language Coding Guidelines
- F. Tool Summary
- G. Number Systems
- H. UML Summary
- I. Java Syntax Summary
- J. HTML Summary

Custom Book and eBook Options

Java Concepts may be ordered in both custom print and eBook formats. You can order a custom print version that includes your choice of chapters—including those from other Horstmann titles. Visit customselect.wiley.com to create your custom order.

Java Concepts is also available in an electronic eBook format with three key advantages:

- The price is significantly lower than for the printed book.
- The eBook contains all material in the printed book plus the web chapters and worked examples in one easy-to-browse format.
- You can customize the eBook to include your choice of chapters.

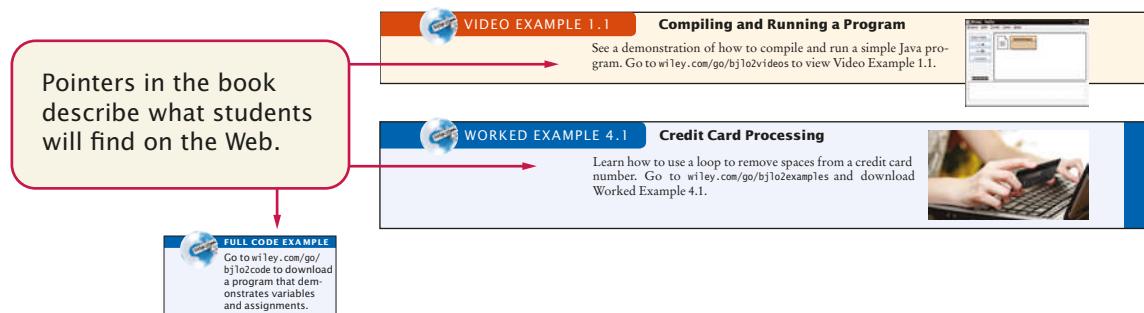
The interactive edition of *Java Concepts* adds even more value by integrating a wealth of interactive exercises into the eBook. See <http://wiley.com/go/bj1o2interactivities> to find out more about this new format.

Please contact your Wiley sales rep for more information about any of these options or check www.wiley.com/college/horstmann for available versions.

Web Resources

This book is complemented by a complete suite of online resources. Go to www.wiley.com/college/horstmann to visit the online companion sites, which include

- Source code for all example programs in the book and its Worked Examples and Video Examples, plus additional example programs.
- Worked Examples that apply the problem-solving steps in the book to other realistic examples.
- Video Examples in which the author explains the steps he is taking and shows his work as he solves a programming problem.
- Lecture presentation slides (for instructors only).
- Solutions to all review and programming exercises (for instructors only).
- A test bank that focuses on skills, not just terminology (for instructors only). This extensive set of multiple-choice questions can be used with a word processor or imported into a course management system.
- “CodeCheck” assignments that allow students to work on programming problems presented in an innovative online service and receive immediate feedback. Instructors can assign exercises that have already been prepared, or easily add their own. Visit <http://codecheck.it> to learn more.



A Walkthrough of the Learning Aids

The pedagogical elements in this book work together to focus on and reinforce key concepts and fundamental principles of programming, with additional tips and detail organized to support and deepen these fundamentals. In addition to traditional features, such as chapter objectives and a wealth of exercises, each chapter contains elements geared to today's visual learner.

Throughout each chapter, **margin notes** show where new concepts are introduced and provide an outline of key ideas.

Additional **full code examples** provides complete programs for students to run and modify.

Annotated **syntax boxes** provide a quick, visual overview of new language constructs.

Annotations explain required components and point to more information on common errors or best practices associated with the syntax.

Like a variable in a computer program, a parking space has an identifier and a contents.

150 Chapter 4 Loops

4.3 The for Loop

The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

It often happens that you want to execute a sequence of statements a given number of times. You can use a while loop that is controlled by a counter, as in the following example:

```
int counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
    System.out.println(counter);
    counter++; // Update the counter
}
```

Because this loop type is so common, there is a special form for it, called the for loop (see Syntax 4.2).

```
for (int counter = 1; counter <= 10; counter++)
{
    System.out.println(counter);
}
```

Some people call this loop *count-controlled*. In contrast, the while loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times; ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.



You can visualize the for loop as an orderly sequence of steps.

Syntax 4.2 for Statement

Syntax

```
for (initialization; condition; update)
{
    statements
}
```

These three expressions should be related.
See Programming Tip 4.1.

This initialization happens once before the loop starts.
The condition is checked before each iteration.
This update is executed after each iteration.

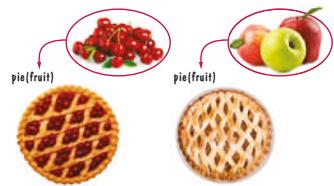
```
for (int i = 5; i <= 10; i++)
{
    sum = sum + i;
}
```

This loop executes 6 times.
See Programming Tip 4.3.



Analogy to everyday objects are used to explain the nature and behavior of concepts such as variables, data types, loops, and more.

Memorable photos reinforce analogies and help students remember the concepts.



A recipe for a fruit pie may say to use any kind of fruit. Here, "fruit" is an example of a parameter variable. Apples and cherries are examples of arguments.

Problem Solving sections teach techniques for generating ideas and evaluating proposed solutions, often using pencil and paper or other artifacts. These sections emphasize that most of the planning and problem solving that makes students successful happens away from the computer.

6.5 Problem Solving: Discovering Algorithms by Manipulating Physical Objects 277

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as Java programmers, we will say that we swap the coins in positions 0 and 4:



Next, we swap the coins in positions 1 and 5:



HOW TO 1.1



Describing an Algorithm with Pseudocode

This is the first of many "How To" sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in Java, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode: a sequence of precise steps formulated in English.

For example, consider this problem: You have the choice of buying two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of \$4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?



Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- `purchase price1` and `fuel efficiency1`, the price and fuel efficiency (in mpg) of the first car
- `purchase price2` and `fuel efficiency2`, the price and fuel efficiency of the second car

We simply want to know which car is the better buy. That is the desired output.

WORKED EXAMPLE 1.1



Writing an Algorithm for Tiling a Floor

This Worked Example shows how to develop an algorithm for laying tile in an alternating pattern of colors.



How To guides give step-by-step guidance for common programming tasks, emphasizing planning and testing. They answer the beginner's question, "Now what do I do?" and integrate key concepts into a problem-solving sequence.

Worked Examples and **Video Examples** apply the steps in the How To to a different example, showing how they can be used to plan, implement, and test a solution to another programming problem.

Table 1 Variable Declarations in Java

Variable Name	Comment
<code>int cans = 6;</code>	Declares an integer variable and initializes it with 6.
<code>int total = cans + bottles;</code>	The initial value need not be a constant. (Of course, cans and bottles must have been previously declared.)
<code>bottles = 1;</code>	Error: The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.1.4.
<code>int bottles = "10";</code>	Error: You cannot initialize a number with a string.
<code>int bottles;</code>	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 37.
<code>int cans, bottles;</code>	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

Example tables support beginners with multiple, concrete examples. These tables point out common errors and present another quick reference to the section's topic.

Figure 3
Execution of
a for Loop

```

❶ Initialize counter      for (int counter = 1; counter <= 10; counter++)
{                           System.out.println(counter);
    counter = 1
}

❷ Check condition        for (int counter = 1; counter <= 10; counter++)
{                           System.out.println(counter);
    counter = 1
}

❸ Execute loop body     for (int counter = 1; counter <= 10; counter++)
{                           System.out.println(counter);
    counter = 1
}

❹ Update counter         for (int counter = 1; counter <= 10; counter++)
{                           System.out.println(counter);
    counter = 2
}

❺ Check condition again for (int counter = 1; counter <= 10; counter++)
{                           System.out.println(counter);
    counter = 2
}

```

Progressive figures trace code segments to help students visualize the program flow. Color is used consistently to make variables and other elements easily recognizable.

```

❶ Method call           result1 = 1
double result1 = cubeVolume(2);

❷ Initializing method parameter variable
result1 = 1
sideLength = 2

❸ About to return to the caller
result1 = 1
sideLength = 2
volume = 8

❹ After method call     result1 = 8
double result1 = cubeVolume(2);

```

Figure 3 Parameter Passing

- The parameter variable `sideLength` of the `cubeVolume` method is created when the method is called. ❶
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, `sideLength` is set to 2. ❷
- The method computes the expression `sideLength * sideLength * sideLength`, which has the value 8. That value is stored in the variable `volume`. ❸
- The method returns. All of its variables are removed. The return value is trans-

Self-check exercises at the end of each section are designed to make students think through the new material—and can spark discussion in lecture.



11. Write the for loop of the `InvestmentTable.java` program as a while loop.

12. How many numbers does this loop print?

```
for (int n = 10; n >= 0; n--)
{
    System.out.println(n);
}
```

13. Write a for loop that prints all even numbers between 10 and 20 (inclusive).

14. Write a for loop that computes the sum of the integers from 1 to n.

15. How would you modify the for loop of the `InvestmentTable.java` program to print all balances until the investment has doubled?

Practice It Now you can try these exercises at the end of the chapter: R4.7, R4.13, E4.8, E4.16.

Optional science and business exercises engage students with realistic applications of Java.

```

sec01/DoubleInvestment.java
1 /**
2  * This program computes the time required to double an investment.
3 */
4 public class DoubleInvestment
5 {
6     public static void main(String[] args)
7     {
8         final double RATE = 5;
9         final double INITIAL_BALANCE = 10000;
10        final double TARGET = 2 * INITIAL_BALANCE;
11
12        double balance = INITIAL_BALANCE;
13        int year = 0;
14
15        // Count the years required for the investment to double
16
17        while (balance < TARGET)
18        {
19            year++;
20            double interest = balance * RATE / 100;
21            balance = balance + interest;
22        }
23
24        System.out.println("The investment doubled after "
25            + year + " years.");
26    }
27 }

```

Science P6.13 Sounds can be represented by an array of “sample values” that describe the intensity of the sound at a point in time. The program `ch06/sound/SoundEffect.java` reads a sound file (in WAV format), calls a method `process` for processing the sample values, and saves the sound file. Your task is to implement the `process` method by introducing an echo. For each



Business P9.6 Implement a superclass `Appointment` and subclasses `OneTime`, `Daily`, and `Monthly`. An appointment has a description (for example, “see the dentist”) and a date and time. Write a method `occursOn(int year, int month, int day)` that checks whether the appointment occurs on that date. For example, for a monthly appointment, you must check whether the day of the month matches. Then fill an array of `Appointment` objects with a mixture of appointments. Have the user enter a date and print out all appointments that occur on that date.



Program listings are carefully designed for easy reading, going well beyond simple color coding. Methods are set off by a subtle outline.

Common Errors describe the kinds of errors that students often make, with an explanation of why the errors occur, and what to do about them.

Common Error 6.4



Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not all consistent. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Data Type	Number of Elements
Array	a.length
Array list	a.size()
String	a.length()

Programming Tip 3.5



Hand-Tracing

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or Java code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

For example, let's trace the getTax method with the data from the program run above.

When the TaxReturn object is constructed, the income instance variable is set to \$80,000 and status is set to MARRIED. Then the getTax method is called. In lines 31 and 32 of TaxReturn.java, tax1 and tax2 are initialized to 0.

```
29 public double getTax()
30 {
31     double tax1 = 0;
32     double tax2 = 0;
33
34     if (status == SINGLE)
35     {
36         if (income <= RATE1_SINGLE_LIMIT)
37             tax1 = RATE1 * income;
38         }
39     else
40     {
41         tax1 = RATE1 * RATE1_SINGLE_LIMIT;
42         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
43     }
}
```



Hand-tracing helps you understand whether a program works correctly.

Programming Tips explain good programming practices, and encourage students to be more productive with tips and techniques such as hand-tracing.

Special Topic 7.2



File Dialog Boxes

In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The JFileChooser class implements a file dialog box for the Swing user-interface toolkit.

The JFileChooser class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple. Construct a file chooser object; then call the showOpenDialog or showSaveDialog method. Both methods show the same dialog box, but the button for selecting a file is labeled "Open" or "Save", depending on which method you call.

For better placement of the dialog box on the screen, you can specify the user-interface component over which to pop up the dialog box. If you don't care where the dialog box pops up, you can simply pass null. The showOpenDialog and showSaveDialog methods return either JFileChooser.APPROVE_OPTION if the user has chosen a file, or JFileChooser.CANCEL_OPTION if the user canceled the selection. If a file was chosen, then you call the getSelectedFile method to obtain a File object that describes the file. Here is a complete example:

```
JFileChooser chooser = new JFileChooser();
```

Special Topics present optional topics and provide additional explanation of others.

Java 8 Note 9.3

8

Lambda Expressions

In the preceding section, you saw how to use interfaces for specifying variations in behavior. The average method needs to measure each object, and it does so by calling the measure method of the supplied Measurer object.

Unfortunately, the caller of the average method has to do a fair amount of work; namely, to define a class that implements the Measurer interface and to construct an object of that class. Java 8 has a convenient shortcut for these steps, provided that the interface has a single abstract method. Such an interface is called a *functional interface* because its purpose is to define a single function. The Measurer interface is an example of a functional interface.

To specify that single function, you can use a *lambda expression*, an expression that defines the parameters and return value of a method in a compact notation. Here is an example:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

This expression defines a function that, given an object, casts it to a BankAccount and returns the balance.



Computing & Society 1.1 Computers Are Everywhere

When computers were first invented in the 1940s, a computer filled an entire room. The photo below shows the ENIAC (electronic numerical integrator and computer), completed in 1946 at the University of Pennsylvania. The ENIAC was used by the military to compute the trajectories of projectiles. Nowadays, computing facilities of search engines, Internet shops, and social networks fill huge buildings called data centers. At the other end of the spectrum, computers are all around us. Your cell phone has a computer inside, as do many credit cards and fare cards for public transit. A modern car has several computers—to control the engine, brakes, lights, and the radio.

The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are now carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies are nowadays often consumed on computers, and computers are almost always involved in their production. The book that you are reading right now could not have been written without computers.



This transit card contains a computer.

Java 8 Notes provide detail about new features in Java 8.

Computing & Society presents social and historical topics on computing—for interest and to fulfill the “historical and social context” requirements of the ACM/IEEE curriculum guidelines.

Acknowledgments

Many thanks to Don Fowley, Graig Donini, Brad Franklin, Dan Sayre, Anna Pham, Laura Abrams, and Billy Ray at John Wiley & Sons for their help with this project. An especially deep acknowledgment and thanks goes to Cindy Johnson for her hard work, sound judgment, and amazing attention to detail.

I am grateful to Jose Cordova, *University of Louisiana at Monroe*, Suzanne Dietrich, *Arizona State University, West Campus*, Byron Hoy, *Stockton University*, Brent Wilson, *George Fox University*, and David Woolbright, *Columbus State University*, for their contributions to the supplemental material.

Every new edition builds on the suggestions and experiences of new and prior reviewers, contributors, and users. Many thanks to the individuals who provided feedback, reviewed the manuscript, made valuable suggestions and contributions, and brought errors and omissions to my attention. They include:

- Lynn Aaron, *SUNY Rockland Community College*
- Karen Arlien, *Bismarck State College*
- Jay Asundi, *University of Texas, Dallas*
- Eugene Backlin, *DePaul University*
- William C. Barge, *Trine University*
- Bruce J. Barton, *Suffolk County Community College*
- Sanjiv K. Bhatia, *University of Missouri, St. Louis*
- Anna Biesczad, *California State University, Channel Islands*
- Jackie Bird, *Northwestern University*
- Eric Bishop, *Northland Pioneer College*
- Paul Bladek, *Edmonds Community College*
- Paul Logasa Bogen II, *Texas A&M University*
- Irene Bruno, *George Mason University*
- Paolo Bucci, *Ohio State University*
- Joe Burgin, *College of Southern Maryland*
- Robert P. Burton, *Brigham Young University*
- Leonello Calabresi, *University of Maryland University College*
- Martine Ceberio, *University of Texas, El Paso*
- Uday Chakraborty, *University of Missouri, St. Louis*
- Suchindran Chatterjee, *Arizona State University*
- Xuemin Chen, *Texas Southern University*
- Haiyan Cheng, *Willamette University*
- Chakib Chraibi, *Barry University*
- Ta-Tao Chuang, *Gonzaga University*
- Vincent Cicirello, *Richard Stockton College*
- Mark Clement, *Brigham Young University*
- Gerald Cohen, *St. Joseph's College*
- Ralph Conrad, *San Antonio College*
- Dave Cook, *Stephen F. Austin State University*
- Rebecca Crellin, *Community College of Allegheny County*
- Leslie Damon, *Vermont Technical College*
- Geoffrey D. Decker, *Northern Illinois University*
- Khaled Deeb, *Barry University, School of Adult and Continuing Education*
- Akshaye Dhawan, *Ursinus College*
- Julius Dichter, *University of Bridgeport*
- Mike Domaratzki, *University of Manitoba*
- Philip Dorin, *Loyola Marymount University*
- Anthony J. Dos Reis, *SUNY New Paltz*
- Elizabeth Drake, *Santa Fe College*
- Tom Duffy, *Norwalk Community College*
- Michael Eckmann, *Skidmore College*
- Sander Eller, *California State Polytechnic University, Pomona*
- Amita Engineer, *Valencia Community College*

- Dave Evans, *Pasadena Community College*
James Factor, *Alverno College*
Chris Fietkiewicz, *Case Western Reserve University*
Terrell Foty, *Portland Community College*
Valerie Frear, *Daytona State College*
Zhenguang Gao, *Framingham State University*
Ryan Garlick, *University of North Texas*
Aaron Garrett, *Jacksonville State University*
Stephen Gilbert, *Orange Coast College*
Rick Giles, *Acadia University*
Peter van der Goes, *Rose State College*
Billie Goldstein, *Temple University*
Michael Gourley, *University of Central Oklahoma*
Grigoriy Grinberg, *Montgomery College*
Linwu Gu, *Indiana University*
Sylvain Guinepain, *University of Oklahoma, Norman*
Bruce Haft, *Glendale Community College*
Nancy Harris, *James Madison University*
Allan M. Hart, *Minnesota State University, Mankato*
Ric Heishman, *George Mason University*
Guy Helmer, *Iowa State University*
Katherin Herbert, *Montclair State University*
Rodney Hoffman, *Occidental College*
May Hou, *Norfolk State University*
John Houlihan, *Loyola University*
Andree Jacobson, *University of New Mexico*
Eric Jiang, *University of San Diego*
Christopher M. Johnson, *Guilford College*
Jonathan Kapleau, *New Jersey Institute of Technology*
Amitava Karmaker, *University of Wisconsin, Stout*
Rajkumar Kempaiah, *College of Mount Saint Vincent*
Mugdha Khaladkar, *New Jersey Institute of Technology*
Richard Kick, *Newbury Park High School*
Julie King, *Sullivan University, Lexington*
Samuel Kohn, *Touro College*
April Kontostathis, *Ursinus College*
Ron Krawitz, *DeVry University*
Nat Kumaresan, *Georgia Perimeter College*
Debbie Lamprecht, *Texas Tech University*
Jian Lin, *Eastern Connecticut State University*
Hunter Lloyd, *Montana State University*
Cheng Luo, *Coppin State University*
Kelvin Lwin, *University of California, Merced*
Frank Malinowski, *Dalton College*
John S. Mallozzi, *Iona College*
Khaled Mansour, *Washtenaw Community College*
Kenneth Martin, *University of North Florida*
Deborah Mathews, *J. Sergeant Reynolds Community College*
Louis Mazzucco, *State University of New York at Cobleskill and Excelsior College*
Drew McDermott, *Yale University*
Patricia McDermott-Wells, *Florida International University*
Hugh McGuire, *Grand Valley State University*
Michael L. Mick, *Purdue University, Calumet*
Jeanne Milostan, *University of California, Merced*
Sandeep Mitra, *SUNY Brockport*
Michel Mitri, *James Madison University*
Kenrick Mock, *University of Alaska Anchorage*
Namdar Mogharreban, *Southern Illinois University*
Jose-Arturo Mora-Soto, *University of Madrid*
Shamsi Moussavi, *Massbay Community College*

- Nannette Napier, *Georgia Gwinnett College*
 Tony Tuan Nguyen, *De Anza College*
 Michael Ondrasek, *Wright State University*
 K. Palaniappan, *University of Missouri*
 James Papademas, *Oakton Community College*
 Gary Parker, *Connecticut College*
 Jody Paul, *Metropolitan State College of Denver*
 Mark Pendergast, *Florida Gulf Coast University*
 James T. Pepe, *Bentley University*
 Jeff Pittges, *Radford University*
 Tom Plunkett, *Virginia Tech*
 Linda L. Preece, *Southern Illinois University*
 Vijay Ramachandran, *Colgate University*
 Craig Reinhart, *California Lutheran University*
 Jonathan Robinson, *Touro College*
 Chaman Lal Sabharwal, *Missouri University of Science & Technology*
 Katherine Salch, *Illinois Central College*
 Namita Sarawagi, *Rhode Island College*
 Ben Schafer, *University of Northern Iowa*
 Walter Schilling, *Milwaukee School of Engineering*
 Jeffrey Paul Scott, *Blackhawk Technical College*
 Amon Seagull, *NOVA Southeastern University*
 Brent Seales, *University of Kentucky*
 Linda Seiter, *John Carroll University*
 Kevin Seppi, *Brigham Young University*
 Ricky J. Sethi, *UCLA, USC ISI, and DeVry University*
 Narasimha Shashidhar, *Sam Houston State University*
 Ali Shaykhian, *Florida Institute of Technology*
 Lal Shimpi, *Saint Augustine's College*
 Victor Shtern, *Boston University*
 Rahul Simha, *George Washington University*
 Jeff Six, *University of Delaware*
 Donald W. Smith, *Columbia College*
 Derek Snow, *University of Southern Alabama*
 Peter Spoerri, *Fairfield University*
 David R. Stampf, *Suffolk County Community College*
 Peter Stanchev, *Kettering University*
 Ryan Stansifer, *Florida Institute of Technology*
 Stu Steiner, *Eastern Washington University*
 Robert Strader, *Stephen F. Austin State University*
 David Stucki, *Otterbein University*
 Ashok Subramanian, *University of Missouri, St Louis*
 Jeremy Suing, *University of Nebraska, Lincoln*
 Dave Sullivan, *Boston University*
 Vaidy Sunderam, *Emory University*
 Hong Sung, *University of Central Oklahoma*
 Monica Sweat, *Georgia Tech University*
 Joseph Szurek, *University of Pittsburgh, Greensburg*
 Jack Tan, *University of Wisconsin*
 Cynthia Tanner, *West Virginia University*
 Russell Tessier, *University of Massachusetts, Amherst*
 Krishnaprasad Thirunarayan, *Wright State University*
 Mark Thomas, *University of Cincinnati Blue Ash*
 Megan Thomas, *California State University, Stanislaus*
 Timothy Urness, *Drake University*
 Eliana Valenzuela-Andrade, *University of Puerto Rico at Arecibo*
 Tammy VanDeGrift, *University of Portland*
 Philip Ventura, *Broward College*
 David R. Vineyard, *Kettering University*
 Qi Wang, *Northwest Vista College*
 Jonathan Weissman, *Finger Lakes Community College*
 Reginald White, *Black Hawk Community College*
 Ying Xie, *Kennesaw State University*
 Arthur Yanushka, *Christian Brothers University*

xviii Acknowledgments

Chen Ye, *University of Illinois, Chicago*
Wook-Sung Yoo, *Fairfield University*
Steve Zale, *Middlesex County College*
Bahram Zartoshty, *California State
University, Northridge*
Frank Zeng, *Indiana Wesleyan
University*

Hairong Zhao, *Purdue University
Calumet*
Stephen Zilora, *Rochester Institute of
Technology*

And a special thank you to our class testers:

Eugene Backlin and the students of DePaul University, Loop
Debra M. Duke and the students of J. Sargeant Reynolds Community College
Gerald Gordon and the students of DePaul University, Loop
Mike Gourley and the students of the University of Central Oklahoma
Mohammad Morovati and the students of the College of DuPage
Mutsumi Nakamura and the students of Arizona State University
George Novacky and the students of the University of Pittsburgh
Darrin Rothe and the students of the Milwaukee School of Engineering
Paige Rutner and the students of Georgia Southern University
Narasimha Shashidhar and the students of Sam Houston State University
Mark Sherriff and the students of the University of Virginia
Frank Zeng and the students of Indiana Wesleyan University

CONTENTS

PREFACE **v**

SPECIAL FEATURES **xxvi**

1 INTRODUCTION **1**

1.1 Computer Programs	2
1.2 The Anatomy of a Computer	3
1.3 The Java Programming Language	6
1.4 Becoming Familiar with Your Programming Environment	7
VE1 Compiling and Running a Program	
1.5 Analyzing Your First Program	11
1.6 Errors	14
1.7 PROBLEM SOLVING Algorithm Design	15
The Algorithm Concept	16
An Algorithm for Solving an Investment Problem	17
Pseudocode	18
From Algorithms to Programs	18
HT1 Describing an Algorithm with Pseudocode	19
WE1 Writing an Algorithm for Tiling a Floor	21
VE2 Dividing Household Expenses	

2 FUNDAMENTAL DATA TYPES **31**

2.1 Variables	32
Variable Declarations	32
Number Types	34
Variable Names	35
The Assignment Statement	36
Constants	37
Comments	37
ST1 Numeric Types in Java	41
ST2 Big Numbers	42
2.2 Arithmetic	43
Arithmetic Operators	43
Increment and Decrement	43
Integer Division and Remainder	44
Powers and Roots	45

Converting Floating-Point Numbers to Integers **46**

J81 Avoiding Negative Remainders **49**

ST3 Combining Assignment and Arithmetic **49**

VE1 Using Integer Division

2.3 Input and Output **50**

Reading Input **50**

Formatted Output **52**

HT1 Carrying Out Computations **56**

WE1 Computing the Cost of Stamps

2.4 PROBLEM SOLVING First Do it By Hand **59**

WE2 Computing Travel Time

2.5 Strings **61**

The String Type **61**

Concatenation **61**

String Input **62**

Escape Sequences **62**

Strings and Characters **63**

Substrings **63**

ST4 Instance Methods and Static Methods **66**

ST5 Using Dialog Boxes for Input and Output **67**

VE2 Computing Distances on Earth

3 DECISIONS **83**

3.1 The if Statement **84**

ST1 The Conditional Operator **89**

3.2 Comparing Numbers and Strings **90**

ST2 Lexicographic Ordering of Strings **94**

HT1 Implementing an if Statement **95**

WE1 Extracting the Middle

3.3 Multiple Alternatives **98**

ST3 The switch Statement **101**

3.4 Nested Branches **102**

ST4 Enumeration Types **107**

VE1 Computing the Plural of an English Word

3.5 PROBLEM SOLVING Flowcharts **107**

3.6 PROBLEM SOLVING Test Cases **110**

ST5 Logging **112**

3.7 Boolean Variables and Operators **113**

xx Contents

ST6 Short-Circuit Evaluation of Boolean Operators 116	5.3 Parameter Passing 217
ST7 De Morgan's Law 117	5.4 Return Values 220
3.8 APPLICATION Input Validation 118	HT1 Implementing a Method 222
VE2 The Genetic Code 	WE1 Generating Random Passwords 
4 LOOPS 142	5.5 Methods Without Return Values 224
4.1 The while Loop 142	5.6 PROBLEM SOLVING Reusable Methods 225
4.2 PROBLEM SOLVING Hand-Tracing 149	5.7 PROBLEM SOLVING Stepwise Refinement 229
4.3 The for Loop 152	5.8 Variable Scope 236
4.4 The do Loop 158	VE1 Debugging 
4.5 APPLICATION Processing Sentinel Values 160	5.9 Recursive Methods (Optional) 240
ST1 The "Loop and a Half" Problem and the break Statement 162	HT2 Thinking Recursively 243
ST2 Redirection of Input and Output 163	VE2 Fully Justified Text 
VE1 Evaluating a Cell Phone Plan 	
4.6 PROBLEM SOLVING Storyboards 164	6 ARRAYS AND ARRAY LISTS 261
4.7 Common Loop Algorithms 167	6.1 Arrays 262
Sum and Average Value 167	Declaring and Using Arrays 262
Counting Matches 167	Array References 265
Finding the First Match 168	Partially Filled Arrays 266
Prompting Until a Match is Found 169	6.2 The Enhanced for Loop 269
Maximum and Minimum 169	6.3 Common Array Algorithms 270
Comparing Adjacent Values 170	Filling 270
HT1 Writing a Loop 171	Sum and Average Value 271
WE1 Credit Card Processing 	Maximum and Minimum 271
4.8 Nested Loops 174	Element Separators 271
WE2 Manipulating the Pixels in an Image 	Linear Search 272
4.9 PROBLEM SOLVING Solve a Simpler Problem First 178	Removing an Element 272
4.10 APPLICATION Random Numbers and Simulations 182	Inserting an Element 273
Generating Random Numbers 182	Swapping Elements 274
Simulating Die Tosses 183	Copying Arrays 275
The Monte Carlo Method 184	Reading Input 276
ST3 Drawing Graphical Shapes 186	ST1 Sorting with the Java Library 279
VE1 Drawing a Spiral 	ST2 Binary Search 279
5 METHODS 211	6.4 Using Arrays with Methods 280
5.1 Methods as Black Boxes 212	ST3 Methods with a Variable Number of Parameters 284
5.2 Implementing Methods 214	6.5 PROBLEM SOLVING Adapting Algorithms 284
	HT1 Working with Arrays 287
	WE1 Rolling the Dice 

6.6 PROBLEM SOLVING	Discovering Algorithms by Manipulating Physical Objects	291	7.3 Command Line Arguments	345
VE1	Removing Duplicates from an Array		HT1	Processing Text Files 348
6.7 Two-Dimensional Arrays	294		WE1	Analyzing Baby Names 
Declaring Two-Dimensional Arrays	295		7.4 Exception Handling	352
Accessing Elements	295		Throwing Exceptions	352
Locating Neighboring Elements	296		Catching Exceptions	354
Computing Row and Column Totals	297		Checked Exceptions	355
Two-Dimensional Array Parameters	298		Closing Resources	357
WE2	A World Population Table 		ST6	Assertions 360
ST4	Two-Dimensional Arrays with Variable Row Lengths 300		ST7	The try/finally Statement 360
ST5	Multidimensional Arrays 301		7.5 APPLICATION	Handling Input Errors 361
6.8 Array Lists	301		VE2	Detecting Accounting Fraud 
Declaring and Using Array Lists	302			
Using the Enhanced for Loop with Array Lists	304			
Copying Array Lists	305			
Array Lists and Methods	305			
Wrappers and Auto-boxing	305			
Using Array Algorithms with Array Lists	307			
Storing Input Values in an Array List	307			
Removing Matches	307			
Choosing Between Array Lists and Arrays	308			
ST6	The Diamond Syntax 311			
VE2	Game of Life 			
7 INPUT/OUTPUT AND EXCEPTION HANDLING 331				
7.1 Reading and Writing Text Files	332			
ST1	Reading Web Pages 335			
ST2	File Dialog Boxes 335			
ST3	Reading and Writing Binary Data 336			
7.2 Text Input and Output	337			
Reading Words	337			
Reading Characters	338			
Classifying Characters	338			
Reading Lines	339			
Scanning a String	340			
Converting Strings to Numbers	340			
Avoiding Errors When Reading Numbers	340			
Mixing Number, Word, and Line Input	341			
Formatting Output	342			
ST4	Regular Expressions 344			
ST5	Reading an Entire File 344			
VE1	Computing a Document's Readability 			
8 OBJECTS AND CLASSES 375				
8.1 Object-Oriented Programming	376			
8.2 Implementing a Simple Class	378			
8.3 Specifying the Public Interface of a Class	381			
ST1	The javadoc Utility 384			
8.4 Designing the Data Representation	385			
8.5 Implementing Instance Methods	386			
8.6 Constructors	389			
ST2	Overloading 393			
8.7 Testing a Class	393			
HT1	Implementing a Class 395			
WE1	Implementing a Menu Class 			
VE1	Paying Off a Loan 			
8.8 Problem Solving: Tracing Objects	399			
8.9 Object References	403			
Shared References	403			
The null Reference	405			
The this Reference	405			
ST3	Calling One Constructor from Another 408			
8.10 Static Variables and Methods	408			
8.11 PROBLEM SOLVING	Patterns for Object Data 410			
Keeping a Total	411			
Counting Events	411			
Collecting Values	412			
Managing Properties of an Object	413			
Modeling Objects with Distinct States	413			
Describing the Position of an Object	414			
VE2	Modeling a Robot Escaping from a Maze 			

8.12 Packages 417

- Organizing Related Classes into Packages 417
 Importing Packages 418
 Package Names 419
 Packages and Source Files 419
ST4 Package Access 421
HT2 Programming with Packages 421

9 INHERITANCE AND INTERFACES 437**9.1 Inheritance Hierarchies 438****9.2 Implementing Subclasses 442****9.3 Overriding Methods 446**

- ST1** Calling the Superclass Constructor 451

9.4 Polymorphism 452

- ST2** Dynamic Method Lookup and the Implicit Parameter 455

- ST3** Abstract Classes 456

- ST4** Final Methods and Classes 457

- ST5** Protected Access 458

- HT1** Developing an Inheritance Hierarchy 458

- WE1** Implementing an Employee Hierarchy for Payroll Processing 

- VE1** Building a Discussion Board 

9.5 Object: The Cosmic Superclass 463

- Overriding the `toString` Method 464

- The `equals` Method 465

- The `instanceof` Operator 466

- ST6** Inheritance and the `toString` Method 468

- ST7** Inheritance and the `equals` Method 469

9.6 Interface Types 470

- Defining an Interface 470

- Implementing an Interface 472

- The Comparable Interface 474

- ST8** Constants in Interfaces 476

- ST9** Generic Interface Types 476

- J81** Static Methods in Interfaces 477

- J82** Default Methods 477

- ST10** Function Objects 478

- J83** Lambda Expressions 479

- WE2** Investigating Number Sequences 

- VE2** Drawing Geometric Shapes 

10 GRAPHICAL USER INTERFACES 493**10.1 Frame Windows 494**

- Displaying a Frame 494

- Adding User-Interface Components to a Frame 495

- Using Inheritance to Customize Frames 497

- ST1** Adding the `main` Method to the Frame Class 498

10.2 Events and Event Handling 498

- Listening to Events 499

- Using Inner Classes for Listeners 501

- APPLICATION** Showing Growth of an Investment 503

- ST2** Local Inner Classes 507

- ST3** Anonymous Inner Classes 508

- J81** Lambda Expressions for Event Handling 509

10.3 Processing Text Input 509

- Text Fields 509

- Text Areas 511

10.4 Creating Drawings 515

- Drawing on a Component 515

- Ovals, Lines, Text, and Color 517

- APPLICATION** Visualizing the Growth of an Investment 520

- HT1** Drawing Graphical Shapes 525

- WE1** Coding a Bar Chart Creator 

- VE1** Solving Crossword Puzzles 

11 ADVANCED USER INTERFACES (WEB ONLY)**11.1 Layout Management****11.2 Choices**

- Radio Buttons

- Check Boxes

- Combo Boxes

- HT1** Laying Out a User Interface

- WE1** Programming a Working Calculator

11.3 Menus**11.4 Exploring the Swing Documentation****11.5 Using Timer Events for Animations**

11.6 Mouse Events**ST1** Keyboard Events**ST2** Event Adapters**WE2** Adding Mouse and Keyboard Support to the Bar Chart Creator**VE1** Designing a Baby Naming Program**12** OBJECT-ORIENTED DESIGN (WEB ONLY) **12.1** Classes and Their Responsibilities

Discovering Classes

The CRC Card Method

Cohesion

12.2 Relationships Between Classes

Dependency

Aggregation

Inheritance

HT1 Using CRC Cards and UML Diagrams in Program Design**ST1** Attributes and Methods in UML Diagrams**ST2** Multiplicities**ST3** Aggregation, Association, and Composition**12.3** APPLICATION Printing an Invoice

Requirements

CRC Cards

UML Diagrams

Method Documentation

Implementation

WE1 Simulating an Automatic Teller Machine**13** RECURSION (WEB ONLY) **13.1** Triangle Numbers**HT1** Thinking Recursively**WE1** Finding Files**13.2** Recursive Helper Methods**13.3** The Efficiency of Recursion**13.4** Permutations**13.5** Mutual Recursion**13.6** Backtracking**WE2** Towers of Hanoi**14** SORTING AND SEARCHING (WEB ONLY) **14.1** Selection Sort**14.2** Profiling the Selection Sort Algorithm**14.3** Analyzing the Performance of the Selection Sort Algorithm**ST1** Oh, Omega, and Theta**ST2** Insertion Sort**14.4** Merge Sort**14.5** Analyzing the Merge Sort Algorithm**ST3** The Quicksort Algorithm**14.6** Searching

Linear Search

Binary Search

14.7 PROBLEM SOLVING Estimating the Running Time of an Algorithm

Linear Time

Quadratic Time

The Triangle Pattern

Logarithmic Time

14.8 Sorting and Searching in the Java Library

Sorting

Binary Search

Comparing Objects

ST4 The Comparator Interface**J81** Comparators with Lambda Expressions**WE1** Enhancing the Insertion Sort Algorithm**15** THE JAVA COLLECTIONS FRAMEWORK (WEB ONLY) **15.1** An Overview of the Collections Framework**15.2** Linked Lists

The Structure of Linked Lists

The `LinkedList` Class of the Java Collections Framework

List Iterators

15.3 Sets

Choosing a Set Implementation

Working with Sets

15.4 Maps

J81 Updating Map Entries

HT1 Choosing a Collection

WE1 Word Frequency

ST1 Hash Functions

15.5 Stacks, Queues, and Priority Queues

Stacks

Queues

Priority Queues

15.6 Stack and Queue Applications

Balancing Parentheses

Evaluating Reverse Polish Expressions

Evaluating Algebraic Expressions

Backtracking

WE2 Simulating a Queue of Waiting Customers

VE1 Building a Table of Contents

ST2 Reverse Polish Notation

APPENDIX A THE BASIC LATIN AND LATIN-1 SUBSETS OF UNICODE **A-1**

APPENDIX B JAVA OPERATOR SUMMARY **A-5**

APPENDIX C JAVA RESERVED WORD SUMMARY **A-7**

APPENDIX D THE JAVA LIBRARY **A-9**

APPENDIX E TOOL SUMMARY 

APPENDIX F NUMBER SYSTEMS 

APPENDIX G UML SUMMARY 

APPENDIX H JAVA SYNTAX SUMMARY 

APPENDIX I HTML SUMMARY 

GLOSSARY **G-1**

INDEX **I-1**

CREDITS **C-1**

ALPHABETICAL LIST OF SYNTAX BOXES

Arrays	263
Array Lists	302
Assignment	36
Cast	46
Catching Exceptions	354
Comparisons	91
Constant Declaration	37
Constructor with Superclass Initializer	452
Constructors	390
for Statement	154
if Statement	86
Input Statement	51
Instance Methods	387
Instance Variable Declaration	379
Interface Types	471
Java Program	12
Package Specification	419
Static Method Declaration	215
Subclass Declaration	444
The Enhanced for Loop	270
The instanceof Operator	467
The throws Clause	357
The try-with-resources Statement	357
Throwing an Exception	352
Two-Dimensional Array Declaration	295
Variable Declaration	33
while Statement	142

CHAPTER	Common Errors	How Tos and Worked Examples
1 Introduction	Omitting Semicolons Misspelling Words	13 15 Compiling and Running a Program Describing an Algorithm with Pseudocode Writing an Algorithm for Tiling a Floor Dividing Household Expenses
2 Fundamental Data Types	Using Undeclared or Uninitialized Variables Overflow Roundoff Errors Unintended Integer Division Unbalanced Parentheses	39 40 40 48 48 Using Integer Division Carrying out Computations Computing the Cost of Stamps Computing Travel Time Computing Distances on Earth
3 Decisions	A Semicolon After the if Condition Exact Comparison of Floating-Point Numbers Using == to Compare Strings The Dangling else Problem Combining Multiple Relational Operators Confusing && and Conditions	88 93 94 106 115 116 Implementing an if Statement Extracting the Middle Computing the Plural of an English Word The Genetic Code
4 Loops	Don't Think "Are We There Yet?" Infinite Loops Off-by-One Errors	146 147 147 Evaluating a Cell Phone Plan Writing a Loop Credit Card Processing Manipulating the Pixels in an Image Drawing a Spiral
5 Methods	Trying to Modify Arguments Missing Return Value	219 222 Implementing a Method Generating Random Passwords Calculating a Course Grade Debugging Thinking Recursively Fully Justified Text



Programming Tips		Special Topics and Java 8 Notes		Computing & Society	
Backup Copies	10			Computers Are Everywhere	5
Choose Descriptive Variable Names	40	Numeric Types in Java	41	The Pentium Floating-Point Bug	50
Do Not Use Magic Numbers	41	Big Numbers	42	International Alphabets and Unicode	68
Spaces in Expressions	49	Avoiding Negative Remainders	49		
Use the API Documentation	55	Combining Assignment and Arithmetic	49		
		Instance Methods and Static Methods	66		
		Using Dialog Boxes for Input and Output	67		
Brace Layout	88	The Conditional Operator	89	The Denver Airport Luggage System	97
Always Use Braces	88	Lexicographic Ordering of Strings	94	Artificial Intelligence	121
Tabs	89	The switch Statement	101		
Avoid Duplication in Branches	90	Enumeration Types	107		
Hand-Tracing	105	Logging	112		
Make a Schedule and Make Time for Unexpected Problems	111	Short-Circuit Evaluation of Boolean Operators	116		
		De Morgan's Law	117		
Use for Loops for Their Intended Purpose Only	157	The Loop-and-a-Half Problem and the break Statement	162	The First Bug	148
Choose Loop Bounds That Match Your Task	157	Redirection of Input and Output	163	Digital Piracy	188
Count Iterations	158	Drawing Graphical Shapes	186		
Flowcharts for Loops	159				
Method Comments	217			Personal Computing	228
Do Not Modify Parameter Variables	219				
Keep Methods Short	234				
Tracing Methods	234				
Stubs	235				
Using a Debugger	239				



CHAPTER	Common Errors	How Tos and Worked Examples
6 Arrays and Array Lists	Bounds Errors 267 Uninitialized Arrays 267 Underestimating the Size of a Data Set 279 Length and Size 311	Working with Arrays 287  Rolling the Dice  Removing Duplicates from an Array Video  A World Population Table Video  Game of Life Video
7 Input/Output and Exception Handling	Backslashes in File Names 335 Constructing a Scanner with a String 335	 Computing a Document's Readability  Processing Text Files 348  Analyzing Baby Names  Detecting Accounting Fraud Video
8 Objects and Classes	Trying to Call a Constructor 392 Declaring a Constructor as void 393 Forgetting to Initialize Object References in a Constructor 407 Confusing Dots 420	 Implementing a Class 395  Implementing a Menu Class  Paying Off a Loan Video  Modeling a Robot Escaping from a Maze Video  Programming with Packages 421
9 Inheritance and Interfaces	Replicating Instance Variables from the Superclass 445 Confusing Super- and Subclasses 446 Accidental Overloading 450 Forgetting to Use super When Invoking a Superclass Method 451 Don't Use Type Tests 468 Forgetting to Declare Implementing Methods as Public 475	 Developing an Inheritance Hierarchy 458  Implementing an Employee Hierarchy for Payroll Processing  Building a Discussion Board Video  Investigating Number Sequences  Drawing Geometric Shapes Video



 Programming Tips		 Special Topics and Java 8 Notes		 Computing & Society	
Use Arrays for Sequences of Related Items Reading Exception Reports	268 286	Sorting with the Java Library Binary Search Methods with a Variable Number of Parameters Two-Dimensional Arrays with Variable Row Lengths Multidimensional Arrays The Diamond Syntax	279 279 284 300 301 311	Computer Viruses	268
Throw Early, Catch Late Do Not Squelch Exceptions Do Throw Specific Exceptions	359 359 360	Reading Web Pages File Dialog Boxes Reading and Writing Binary Data Regular Expressions Reading an Entire File Assertions The try/finally Statement	335 335 336 344 344 360 360	Encryption Algorithms The Ariane Rocket Incident	351 361
All Instance Variables Should Be Private; Most Methods Should Be Public	388	The javadoc Utility Overloading Calling One Constructor from Another Package Access	384 393 408 421	Open Source and Free Software Electronic Voting Machines	402 416
Use a Single Class for Variation in Values, Inheritance for Variation in Behavior Comparing Integers and Floating-Point Numbers	442 475	Calling the Superclass Constructor Dynamic Method Lookup and the Implicit Parameter Abstract Classes Final Methods and Classes Protected Access Inheritance and the <code>toString</code> Method Inheritance and the <code>equals</code> Method Constants in Interfaces Generic Interface Types • Static Methods in Interfaces • Default Methods Function Objects • Lambda Expressions	451 455 456 457 458 468 469 476 476 477 477 478 479	Who Controls the Internet?	481



CHAPTER

Common
ErrorsHow Tos
and
Worked Examples**10** Graphical User Interfaces

- Modifying Parameter Types in the Implementing Method 506
- Forgetting to Attach a Listener 506
- Forgetting to Repaint 524
- By Default, Components Have Zero Width and Height 525

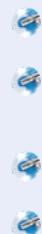
- Drawing Graphical Shapes
- Coding a Bar Chart Creator
- Solving Crossword Puzzles

525
 Video

11 Advanced User Interfaces

(WEB ONLY)

- Laying Out a User Interface
- Programming a Working Calculator
- Adding Mouse and Keyboard Support to the Bar Chart Creator
- Designing a Baby Naming Program

**12** Object-Oriented Design

(WEB ONLY)

- Using CRC Cards and UML Diagrams in Program Design
- Simulating an Automatic Teller Machine

**13** Recursion

(WEB ONLY)

- Infinite Recursion
- Tracing Through Recursive Methods



- Thinking Recursively
- Finding Files
- Towers of Hanoi

**14** Sorting and Searching

(WEB ONLY)

- The `compareTo` Method Can Return Any Integer, Not Just -1, 0, and 1



- Enhancing the Insertion Sort Algorithm

**15** The Java Collections Framework

(WEB ONLY)

- Choosing a Collection
- Word Frequency
- Simulating a Queue of Waiting Customers
- Building a Table of Contents



 Programming Tips	 Special Topics and Java 8 Notes	 Computing & Society
Don't Use a Frame as a Listener 506	Adding the main Method to the Frame Class 498 Local Inner Classes 507 Anonymous Inner Classes 508  Lambda Expressions for Event Handling 509	
Use a GUI Builder	 Keyboard Events  Event Adapters	
Make Parallel Arrays into Arrays of Objects	 Attributes and Methods in UML Diagrams  Multiplicities  Aggregation, Association, and Composition	 Databases and Privacy
		The Limits of Computation 
	 Oh, Omega, and Theta  Insertion Sort  The Quicksort Algorithm  The Comparator Interface  Comparators with Lambda Expressions	The First Programmer 
Use Interface References to Manipulate Data Structures	 Updating Map Entries  Hash Functions  Reverse Polish Notation	Standardization 

INTRODUCTION

CHAPTER GOALS

To learn about computers
and programming

To compile and run your first Java program

To recognize compile-time and run-time errors

To describe an algorithm with pseudocode



© JanPietruszka/iStockphoto.

CHAPTER CONTENTS

1.1 COMPUTER PROGRAMS 2

1.2 THE ANATOMY OF A COMPUTER 3

C&S Computers Are Everywhere 5

1.3 THE JAVA PROGRAMMING LANGUAGE 6

1.4 BECOMING FAMILIAR WITH YOUR PROGRAMMING ENVIRONMENT 7

PT1 Backup Copies 10

VE1 Compiling and Running a Program

1.5 ANALYZING YOUR FIRST PROGRAM 11

SYN Java Program 12

CE1 Omitting Semicolons 13

1.6 ERRORS 14

CE2 Misspelling Words 15

1.7 PROBLEM SOLVING: ALGORITHM DESIGN 15

HT1 Describing an Algorithm with Pseudocode 19

WE1 Writing an Algorithm for Tiling a Floor 21

VE2 Dividing Household Expenses



© JanPietruszka/iStockphoto.

Just as you gather tools, study a project, and make a plan for tackling it, in this chapter you will gather up the basics you need to start learning to program. After a brief introduction to computer hardware, software, and programming in general, you will learn how to write and run your first Java program. You will also learn how to diagnose and fix programming errors, and how to use pseudocode to describe an algorithm—a step-by-step description of how to solve a problem—as you plan your computer programs.

1.1 Computer Programs

Computers execute very basic instructions in rapid succession.

A computer program is a sequence of instructions and decisions.

Programming is the act of designing and implementing computer programs.

You have probably used a computer for work or fun. Many people use computers for everyday tasks such as electronic banking or writing a term paper. Computers are good for such tasks. They can handle repetitive chores, such as totaling up numbers or placing words on a page, without getting bored or exhausted.

The flexibility of a computer is quite an amazing phenomenon. The same machine can balance your checkbook, lay out your term paper, and play a game. In contrast, other machines carry out a much narrower range of tasks; a car drives and a toaster toasts. Computers can carry out a wide range of tasks because they execute different programs, each of which directs the computer to work on a specific task.

The computer itself is a machine that stores data (numbers, words, pictures), interacts with devices (the monitor, the sound system, the printer), and executes programs. A **computer program** tells a computer, in minute detail, the sequence of steps that are needed to fulfill a task. The physical computer and peripheral devices are collectively called the **hardware**. The programs the computer executes are called the **software**.

Today's computer programs are so sophisticated that it is hard to believe that they are composed of extremely primitive instructions. A typical instruction may be one of the following:

- Put a red dot at a given screen position.
- Add up two numbers.
- If this value is negative, continue the program at a certain instruction.

The computer user has the illusion of smooth interaction because a program contains a huge number of such instructions, and because the computer can execute them at great speed.

The act of designing and implementing computer programs is called **programming**. In this book, you will learn how to program a computer—that is, how to direct the computer to execute tasks.

To write a computer game with motion and sound effects or a word processor that supports fancy fonts and pictures is a complex task that requires a team of many highly-skilled programmers. Your first programming efforts will be more mundane. The concepts and skills you learn in this book form an important foundation, and you should not be disappointed if your first programs do not rival the sophisticated software that is familiar to you. Actually, you will find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer precisely and quickly carry out a task that would take you hours of drudgery, to

make small changes in a program that lead to immediate improvements, and to see the computer become an extension of your mental powers.

SELF CHECK


1. What is required to play music on a computer?
2. Why is a CD player less flexible than a computer?
3. What does a computer user need to know about programming in order to play a video game?

1.2 The Anatomy of a Computer

To understand the programming process, you need to have a rudimentary understanding of the building blocks that make up a computer. We will look at a personal computer. Larger computers have faster, larger, or more powerful components, but they have fundamentally the same design.

At the heart of the computer lies the **central processing unit (CPU)** (see Figure 1). The inside wiring of the CPU is enormously complicated. For example, the Intel Core processor (a popular CPU for personal computers at the time of this writing) is composed of several hundred million structural elements, called *transistors*.

The CPU performs program control and data processing. That is, the CPU locates and executes the program instructions; it carries out arithmetic operations such as addition, subtraction, multiplication, and division; it fetches data from external memory or devices and places processed data into storage.

There are two kinds of storage. Primary storage, or memory, is made from electronic circuits that can store data, provided they are supplied with electric power. **Secondary storage**, usually a **hard disk** (see Figure 2)

The central processing unit (CPU) performs program control and data processing.

Storage devices include memory and secondary storage.

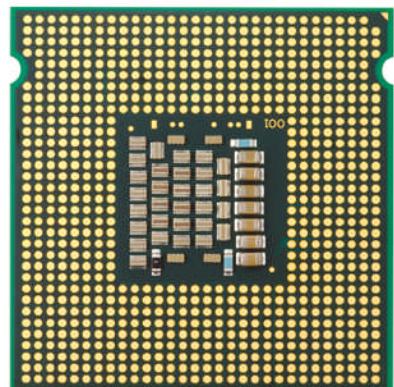


Figure 1 Central Processing Unit

© Amorphis/Stockphoto.



PhotoDisc, Inc./Getty Images, Inc.

Figure 2 A Hard Disk

4 Chapter 1 Introduction

or a solid-state drive, provides slower and less expensive storage that persists without electricity. A hard disk consists of rotating platters, which are coated with a magnetic material. A solid-state drive uses electronic components that can retain information without power, and without moving parts.

To interact with a human user, a computer requires peripheral devices. The computer transmits information (called *output*) to the user through a display screen, speakers, and printers. The user can enter information (called *input*) for the computer by using a keyboard or a pointing device such as a mouse.

Some computers are self-contained units, whereas others are interconnected through **networks**. Through the network cabling, the computer can read data and programs from central storage locations or send data to other computers. To the user of a networked computer, it may not even be obvious which data reside on the computer itself and which are transmitted through the network.

Figure 3 gives a schematic overview of the architecture of a personal computer. Program instructions and data (such as text, numbers, audio, or video) reside in secondary storage or elsewhere on the network. When a program is started, its instructions are brought into memory, where the CPU can read them. The CPU reads and executes one instruction at a time. As directed by these instructions, the CPU reads data, modifies it, and writes it back to memory or secondary storage. Some program instructions will cause the CPU to place dots on the display screen or printer or to vibrate the speaker. As these actions happen many times over and at great speed, the human user will perceive images and sound. Some program instructions read user input from the keyboard, mouse, touch sensor, or microphone. The program analyzes the nature of these inputs and then executes the next appropriate instruction.

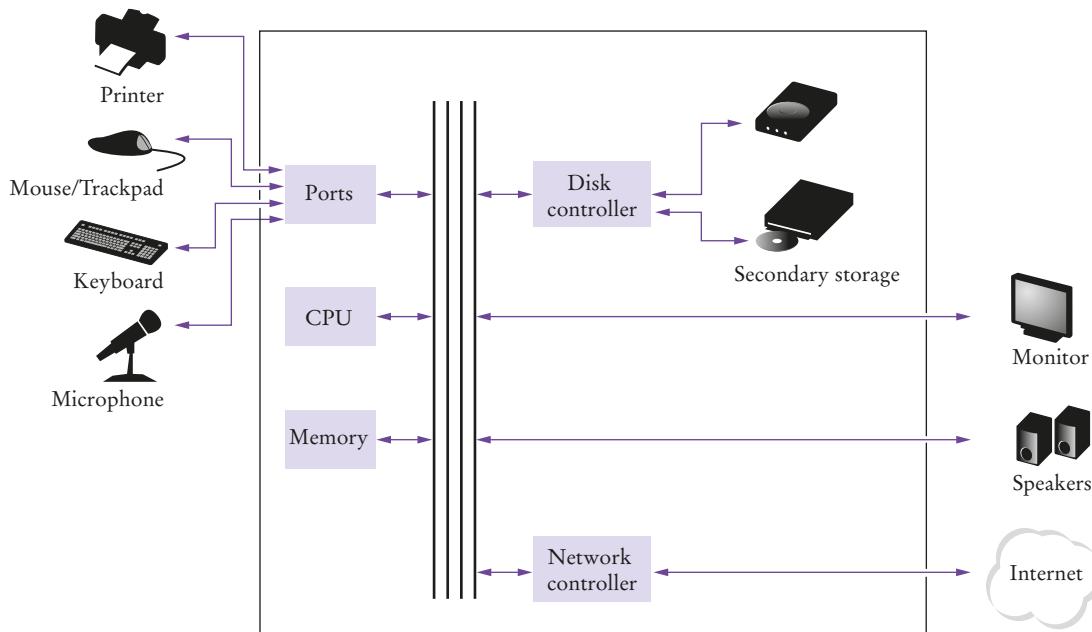


Figure 3 Schematic Design of a Personal Computer



4. Where is a program stored when it is not currently running?
5. Which part of the computer carries out arithmetic operations, such as addition and multiplication?
6. A modern smartphone is a computer, comparable to a desktop computer. Which components of a smartphone correspond to those shown in Figure 3?

Practice It Now you can try these exercises at the end of the chapter: R1.2, R1.3.



Computing & Society 1.1 Computers Are Everywhere

When computers were first invented in the 1940s, a computer filled an entire room. The photo below shows the ENIAC (electronic numerical integrator and computer), completed in 1946 at the University of Pennsylvania. The ENIAC was used by the military to compute the trajectories of projectiles. Nowadays, computing facilities of search engines, Internet shops, and social networks fill huge buildings called data centers. At the other end of the spectrum, computers are all around us. Your cell phone has a computer inside, as do many credit cards and fare cards for public transit. A modern car has several computers—to control the engine, brakes, lights, and the radio.

The advent of ubiquitous computing changed many aspects of our lives. Factories used to employ people to do repetitive assembly tasks that are today carried out by computer-controlled robots, operated by a few people who know how to work with those computers. Books, music, and movies nowadays are often consumed on computers, and computers are almost always involved in their production. The book that you are reading right now could not have



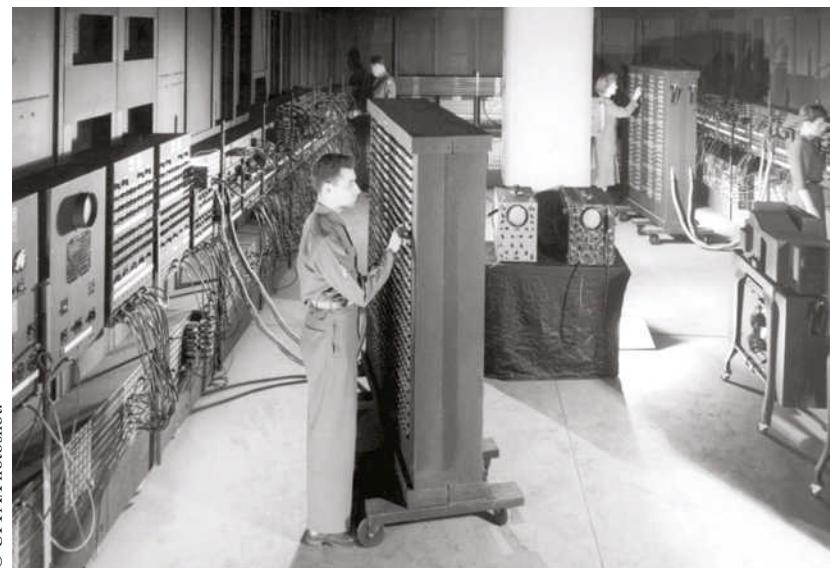
© Maurice Savage/Alamy Limited.

This transit card contains a computer.

been written without computers.

Knowing about computers and how to program them has become an essential skill in many careers. Engineers design computer-controlled cars and medical equipment that preserve lives. Computer scientists develop programs that help people come together to support social causes. For example, activists used social networks to share videos showing abuse by repressive regimes, and this information was instrumental in changing public opinion.

As computers, large and small, become ever more embedded in our everyday lives, it is increasingly important for everyone to understand how they work, and how to work with them. As you use this book to learn how to program a computer, you will develop a good understanding of computing fundamentals that will make you a more informed citizen and, perhaps, a computing professional.



© UPPA/Photoshot.

The ENIAC

1.3 The Java Programming Language

James Sullivan/Getty Images



James Gosling

Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.

Java was designed to be safe and portable, benefiting both Internet users and students.

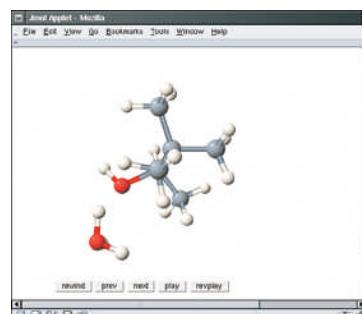
In order to write a computer program, you need to provide a sequence of instructions that the CPU can execute. A computer program consists of a large number of simple CPU instructions, and it is tedious and error-prone to specify them one by one. For that reason, **high-level programming languages** have been created. In a high-level language, you specify the actions that your program should carry out. A **compiler** translates the high-level instructions into the more detailed instructions (called **machine code**) required by the CPU. Many different programming languages have been designed for different purposes.

In 1991, a group led by James Gosling and Patrick Naughton at Sun Microsystems designed a programming language, code-named “Green”, for use in consumer devices, such as intelligent television “set-top” boxes. The language was designed to be simple, secure, and usable for many different processor types. No customer was ever found for this technology.

Gosling recounts that in 1994 the team realized, “We could write a really cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we’d done: architecture neutral, real-time, reliable, secure.” Java was introduced to an enthusiastic crowd at the SunWorld exhibition in 1995, together with a browser that ran **applets**—Java code that can be located anywhere on the Internet. The figure at right shows a typical example of an applet.

Since then, Java has grown at a phenomenal rate. Programmers have embraced the language because it is easier to use than its closest rival, C++. In addition, Java has a rich **library** that makes it possible to write portable programs that can bypass proprietary operating systems—a feature that was eagerly sought by those who wanted to be independent of those proprietary systems and was bitterly fought by their vendors. A “micro edition” and an “enterprise edition” of the Java library allow Java programmers to target hardware ranging from smart cards to the largest Internet servers.

Because Java was designed for the Internet, it has two attributes that make it very suitable for beginners: safety and portability.



An Applet for Visualizing Molecules

Table 1 Java Versions (since Version 1.0 in 1996)

Version	Year	Important New Features	Version	Year	Important New Features
1.1	1997	Inner classes	5	2004	Generic classes, enhanced for loop, auto-boxing, enumerations, annotations
1.2	1998	Swing, Collections framework	6	2006	Library improvements
1.3	2000	Performance enhancements	7	2011	Small language changes and library improvements
1.4	2002	Assertions, XML support	8	2014	Function expressions, streams, new date/time library

Java was designed so that anyone can execute programs in their browser without fear. The safety features of the Java language ensure that a program is terminated if it tries to do something unsafe. Having a safe environment is also helpful for anyone learning Java. When you make an error that results in unsafe behavior, your program is terminated and you receive an accurate error report.

The other benefit of Java is portability. The same Java program will run, without change, on Windows, UNIX, Linux, or Macintosh. In order to achieve portability, the Java compiler does not translate Java programs directly into CPU instructions. Instead, compiled Java programs contain instructions for the **Java virtual machine**, a program that simulates a real CPU. Portability is another benefit for the beginning student. You do not have to learn how to write programs for different platforms.

At this time, Java is firmly established as one of the most important languages for general-purpose programming as well as for computer science instruction. However, although Java is a good language for beginners, it is not perfect, for three reasons.

Because Java was not specifically designed for students, no thought was given to making it really simple to write basic programs. A certain amount of technical machinery is necessary to write even the simplest programs. This is not a problem for professional programmers, but it can be a nuisance for beginning students. As you learn how to program in Java, there will be times when you will be asked to be satisfied with a preliminary explanation and wait for more complete detail in a later chapter.

Java has been extended many times during its life—see Table 1. In this book, we assume that you have Java version 7 or later.

Finally, you cannot hope to learn all of Java in one course. The Java language itself is relatively simple, but Java contains a vast set of *library packages* that are required to write useful programs. There are packages for graphics, user-interface design, cryptography, networking, sound, database storage, and many other purposes. Even expert Java programmers cannot hope to know the contents of all of the packages—they just use those that they need for particular projects.

Using this book, you should expect to learn a good deal about the Java language and about the most important packages. Keep in mind that the central goal of this book is not to make you memorize Java minutiae, but to teach you how to think about programming.

SELF CHECK



7. What are the two most important benefits of the Java language?
8. How long does it take to learn the entire Java library?

Practice It

Now you can try this exercise at the end of the chapter: R1.5.

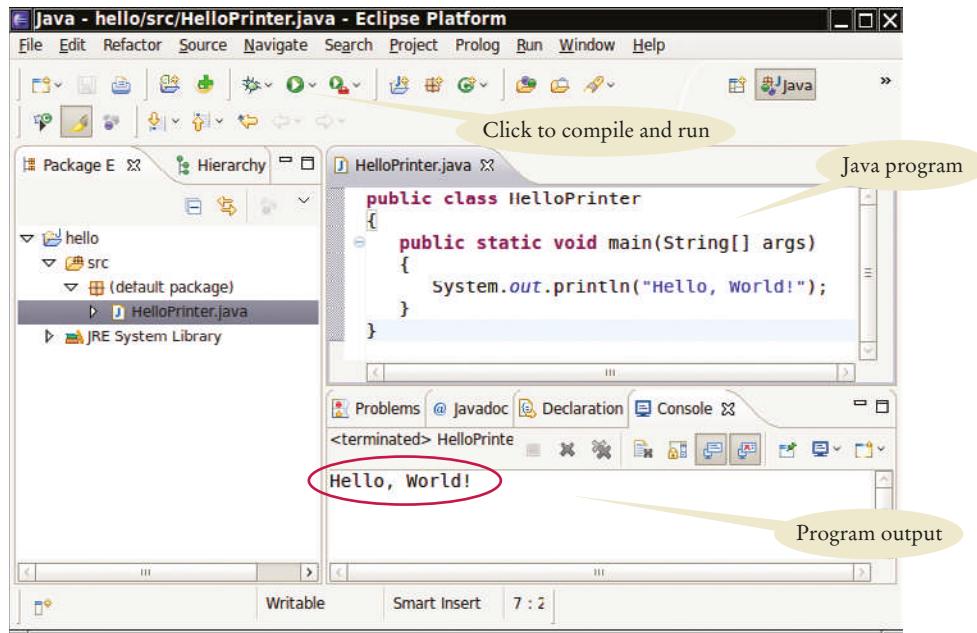
1.4 Becoming Familiar with Your Programming Environment

Set aside time to become familiar with the programming environment that you will use for your class work.

Many students find that the tools they need as programmers are very different from the software with which they are familiar. You should spend some time making yourself familiar with your programming environment. Because computer systems vary widely, this book can only give an outline of the steps you need to follow. It is a good idea to participate in a hands-on lab, or to ask a knowledgeable friend to give you a tour.

8 Chapter 1 Introduction

Figure 4
Running the HelloPrinter Program in an Integrated Development Environment



An editor is a program for entering and modifying text, such as a Java program.

Step 1 Start the Java development environment.

Computer systems differ greatly in this regard. On many computers there is an **integrated development environment** in which you can write and test your programs. On other computers you first launch an **editor**, a program that functions like a word processor, in which you can enter your Java instructions; you then open a **console window** and type commands to execute your program. You need to find out how to get started with your environment.

Step 2 Write a simple program.

The traditional choice for the very first program in a new programming language is a program that displays a simple greeting: “Hello, World!”. Let us follow that tradition. Here is the “Hello, World!” program in Java:

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

We will examine this program in the next section.

No matter which programming environment you use, you begin your activity by typing the program statements into an editor window.

Create a new file and call it `HelloPrinter.java`, using the steps that are appropriate for your environment. (If your environment requires that you supply a project name in addition to the file name, use the name `hello` for the project.) Enter the program instructions *exactly* as they are given above. Alternatively, locate the electronic copy in this book’s companion code and paste it into your editor.

Figure 5
Running the HelloPrinter Program in a Console Window

```

Terminal
File Edit View Terminal Tabs Help
-$ cd BigJava/ch01/hello
~/BigJava/ch01/hello$ javac HelloPrinter.java
~/BigJava/ch01/hello$ java HelloPrinter
Hello, World!
~/BigJava/ch01/hello$ 

```

Java is case sensitive.
You must be careful
about distinguishing
between upper- and
lowercase letters.

As you write this program, pay careful attention to the various symbols, and keep in mind that Java is **case sensitive**. You must enter upper- and lowercase letters exactly as they appear in the program listing. You cannot type `MAIN` or `PrintLn`. If you are not careful, you will run into problems—see Common Error 1.2 on page 15.

Step 3 Run the program.

The process for running a program depends greatly on your programming environment. You may have to click a button or enter some commands. When you run the test program, the message

Hello, World!

will appear somewhere on the screen (see Figures 4 and 5).

The Java compiler translates source code into class files that contain instructions for the Java virtual machine.

In order to run your program, the Java compiler translates your **source files** (that is, the statements that you wrote) into *class files*. (A class file contains instructions for the Java virtual machine.) After the compiler has translated your **source code** into virtual machine instructions, the virtual machine executes them. During execution, the virtual machine accesses a library of pre-written code, including the implementations of the `System` and `PrintStream` classes that are necessary for displaying the program's output. Figure 6 summarizes the process of creating and running a Java program. In some programming environments, the compiler and virtual machine are essentially invisible to the programmer—they are automatically executed whenever you ask to run a Java program. In other environments, you need to launch the compiler and virtual machine explicitly.

Step 4 Organize your work.

As a programmer, you write programs, try them out, and improve them. You store your programs in **files**. Files are stored in **folders** or **directories**. A folder can contain

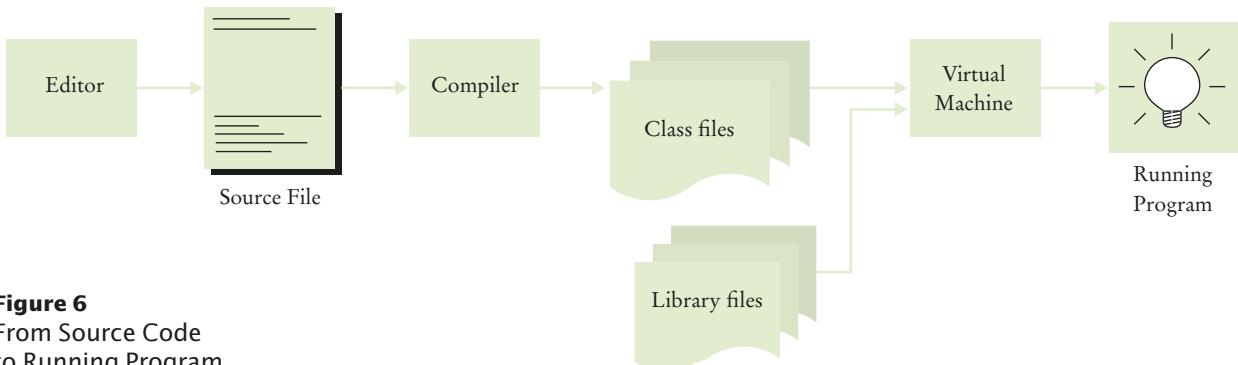


Figure 6
From Source Code
to Running Program



Figure 7
A Folder Hierarchy

files as well as other folders, which themselves can contain more files and folders (see Figure 7). This hierarchy can be quite large, and you need not be concerned with all of its branches. However, you should create folders for organizing your work. It is a good idea to make a separate folder for your programming coursework. Inside that folder, make a separate folder for each program.

Some programming environments place your programs into a default location if you don't specify a folder yourself. In that case, you need to find out where those files are located.

Be sure that you understand where your files are located in the folder hierarchy. This information is essential when you submit files for grading, and for making backup copies (see Programming Tip 1.1).



9. Where is the `HelloPrinter.java` file stored on your computer?
10. What do you do to protect yourself from data loss when you work on programming projects?

Practice It

Now you can try this exercise at the end of the chapter: R1.6.

Programming Tip 1.1



Backup Copies

You will spend many hours creating and improving Java programs. It is easy to delete a file by accident, and occasionally files are lost because of a computer malfunction. Retyping the contents of lost files is frustrating and time-consuming. It is therefore crucially important that you learn how to safeguard files and get in the habit of doing so *before* disaster strikes. Backing up files on a memory stick is an easy and convenient storage method for many people. Another increasingly popular form of backup is Internet file storage. Here are a few pointers to keep in mind:



© Tatiana Popova/iStockphoto.

Develop a strategy for keeping backup copies of your work before disaster strikes.

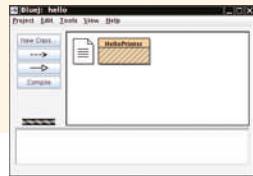
- *Back up often.* Backing up a file takes only a few seconds, and you will hate yourself if you have to spend many hours recreating work that you could have saved easily. I recommend that you back up your work once every thirty minutes.
- *Rotate backups.* Use more than one directory for backups, and rotate them. That is, first back up onto the first directory. Then back up onto the second directory. Then use the third, and then go back to the first. That way you always have three recent backups. If your recent changes made matters worse, you can then go back to the older version.
- *Pay attention to the backup direction.* Backing up involves copying files from one place to another. It is important that you do this right—that is, copy from your work location to the backup location. If you do it the wrong way, you will overwrite a newer file with an older version.
- *Check your backups once in a while.* Double-check that your backups are where you think they are. There is nothing more frustrating than to find out that the backups are not there when you need them.
- *Relax, then restore.* When you lose a file and need to restore it from a backup, you are likely to be in an unhappy, nervous state. Take a deep breath and think through the recovery process before you start. It is not uncommon for an agitated computer user to wipe out the last backup when trying to restore a damaged file.



VIDEO EXAMPLE 1.1

Compiling and Running a Program

See a demonstration of how to compile and run a simple Java program. Go to wiley.com/go/bj102videos to view Video Example 1.1.



1.5 Analyzing Your First Program

© Amanda Rohde/iStockphoto.



In this section, we will analyze the first Java program in detail. Here again is the source code:

sec04/HelloPrinter.java

```

1  public class HelloPrinter
2  {
3      public static void main(String[] args)
4      {
5          // Display a greeting in the console window
6          System.out.println("Hello, World!");
7      }
8  }
9 }
```

The line

```
public class HelloPrinter
```

indicates the declaration of a **class** called `HelloPrinter`.

Every Java program consists of one or more classes. Classes are the fundamental building blocks of Java programs. You will have to wait until Chapter 8 for a full explanation of classes.

The word **public** denotes that the class is usable by the “public”. You will later encounter **private** features.

In Java, every source file can contain at most one public class, and the name of the public class must match the name of the file containing the class. For example, the class `HelloPrinter` must be contained in a file named `HelloPrinter.java`.

The construction

```
public static void main(String[] args)
{
    ...
}
```

declares a **method** called `main`. A method contains a collection of programming instructions that describe how to carry out a particular task. Every Java application must have a **main method**. Most Java programs contain other methods besides `main`, and you will see in Chapter 5 how to write other methods.

The term **static** is explained in more detail in Chapter 8, and the meaning of `String[] args` is covered in Chapter 7. At this time, simply consider

```
public class ClassName
{
    public static void main(String[] args)
    {
        ...
    }
}
```

Classes are the fundamental building blocks of Java programs.

Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.

Each class contains declarations of methods. Each method contains a sequence of instructions.

as a part of the “plumbing” that is required to create a Java program. Our first program has all instructions inside the `main` method of the class.

The `main` method contains one or more instructions called **statements**. Each statement ends in a semicolon (`;`). When a program runs, the statements in the `main` method are executed one by one.

In our example program, the `main` method has a single statement:

```
System.out.println("Hello, World!");
```

This statement prints a line of text, namely “Hello, World!”. In this statement, we *call* a method which, for reasons that we will not explain here, is specified by the rather long name `System.out.println`.

We do not have to implement this method—the programmers who wrote the Java library already did that for us. We simply want the method to perform its intended task, namely to print a value.

Whenever you call a method in Java, you need to specify

1. The method you want to use (in this case, `System.out.println`).
2. Any values the method needs to carry out its task (in this case, “Hello, World!”). The technical term for such a value is an **argument**. Arguments are enclosed in parentheses. Multiple arguments are separated by commas.

A sequence of characters enclosed in quotation marks

```
"Hello, World!"
```

A method is called by specifying the method and its arguments.

A string is a sequence of characters enclosed in quotation marks.

is called a **string**. You must enclose the contents of the string inside quotation marks so that the compiler knows you literally mean “Hello, World!”. There is a reason for this requirement. Suppose you need to print the word *main*. By enclosing it in quotation marks, “main”, the compiler knows you mean the sequence of characters `m a i n`, not the method named `main`. The rule is simply that you must enclose all text strings in quotation marks, so that the compiler considers them plain text and does not try to interpret them as program instructions.

Syntax 1.1 Java Program

Every Java program contains a main method with this header.

The statements inside the main method are executed when the program runs.

Every program contains at least one class. Choose a class name that describes the program action.

```
public class HelloPrinter
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Be sure to match the opening and closing braces.

Each statement ends in a semicolon. See Common Error 1.1.

Replace this statement when you write your own programs.

You can also print numerical values. For example, the statement

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

evaluates the expression $3 + 4$ and displays the number 7.

The `System.out.println` method prints a string or a number and then starts a new line. For example, the sequence of statements

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

prints two lines of text:

```
Hello
World!
```

There is a second method, `System.out.print`, that you can use to print an item without starting a new line. For example, the output of the two statements

```
System.out.print("00");
System.out.println(3 + 4);
```

is the single line

```
007
```

SELF CHECK



11. How do you modify the `HelloPrinter` program to greet you instead?
12. How would you modify the `HelloPrinter` program to print the word “Hello” vertically?
13. Would the program continue to work if you replaced line 7 with this statement?
`System.out.println(Hello);`
14. What does the following set of statements print?
`System.out.print("My lucky number is");
System.out.println(3 + 4 + 5);`
15. What do the following statements print?
`System.out.println("Hello");
System.out.println("");
System.out.println("World");`

Practice It

Now you can try these exercises at the end of the chapter: R1.7, R1.8, E1.5, E1.8.

Common Error 1.1



Omitting Semicolons

In Java every statement must end in a semicolon. Forgetting to type a semicolon is a common error. It confuses the compiler, because the compiler uses the semicolon to find where one statement ends and the next one starts. The compiler does not use line breaks or closing braces to recognize the end of statements. For example, the compiler considers

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

a single statement, as if you had written

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

Then it doesn’t understand that statement, because it does not expect the word `System` following the closing parenthesis after “Hello”.

The remedy is simple. Scan every statement for a terminating semicolon, just as you would check that every English sentence ends in a period. However, do not add a semicolon at the end of `public class Hello` or `public static void main`. These lines are not statements.

1.6 Errors

A compile-time error is a violation of the programming language rules that is detected by the compiler.

Experiment a little with the `HelloPrinter` program. What happens if you make a typing error such as

```
System.ou.println("Hello, World!");
System.out.println("Hello, Word!");
```

In the first case, the compiler will complain. It will say that it has no clue what you mean by `ou`. The exact wording of the error message is dependent on your development environment, but it might be something like “Cannot find symbol `ou`”. This is a **compile-time error**. Something is wrong according to the rules of the language and the compiler finds it. For this reason, compile-time errors are often called **syntax errors**. When the compiler finds one or more errors, it refuses to translate the program into Java virtual machine instructions, and as a consequence you have no program that you can run. You must fix the error and compile again. In fact, the compiler is quite picky, and it is common to go through several rounds of fixing compile-time errors before compilation succeeds for the first time.

If the compiler finds an error, it will not simply stop and give up. It will try to report as many errors as it can find, so you can fix them all at once.

Sometimes, an error throws the compiler off track. Suppose, for example, you forget the quotation marks around a string: `System.out.println(Hello, World!)`. The compiler will not complain about the missing quotation marks. Instead, it will report “Cannot find symbol `Hello`”. Unfortunately, the compiler is not very smart and it does not realize that you meant to use a string. It is up to you to realize that you need to enclose strings in quotation marks.

The error in the second line above is of a different kind. The program will compile and run, but its output will be wrong. It will print

```
Hello, Word!
```

A run-time error causes a program to take an action that the programmer did not intend.

This is a **run-time error**. The program is syntactically correct and does something, but it doesn’t do what it is supposed to do. Because run-time errors are caused by logical flaws in the program, they are often called **logic errors**.

This particular run-time error did not include an error message. It simply produced the wrong output. Some kinds of run-time errors are so severe that they generate an **exception**: an error message from the Java virtual machine. For example, if your program includes the statement

```
System.out.println(1 / 0);
```

you will get a run-time error message “Division by zero”.

During program development, errors are unavoidable. Once a program is longer than a few lines, it would require superhuman concentration to enter it correctly without slipping up once. You will find yourself omitting semicolons or quotation marks more often than you would like, but the compiler will track down these problems for you.

Run-time errors are more troublesome. The compiler will not find them—in fact, the compiler will cheerfully translate any program as long as its syntax is correct—



© Martin Carlsson/iStockphoto.

Programmers spend a fair amount of time fixing compile-time and run-time errors.



FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download three programs that illustrate errors.

but the resulting program will do something wrong. It is the responsibility of the program author to test the program and find any run-time errors.

SELF CHECK



- 16.** Suppose you omit the "" characters around `Hello, World!` from the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
- 17.** Suppose you change `println` to `printline` in the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
- 18.** Suppose you change `main` to `hello` in the `HelloPrinter.java` program. Is this a compile-time error or a run-time error?
- 19.** When you used your computer, you may have experienced a program that “crashed” (quit spontaneously) or “hung” (failed to respond to your input). Is that behavior a compile-time error or a run-time error?
- 20.** Why can't you test a program for run-time errors when it has compiler errors?

Practice It

Now you can try these exercises at the end of the chapter: R1.9, R1.10, R1.11.

Common Error 1.2



Misspelling Words

If you accidentally misspell a word, then strange things may happen, and it may not always be completely obvious from the error messages what went wrong. Here is a good example of how simple spelling errors can cause trouble:

```
public class HelloPrinter
{
    public static void Main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

This class declares a method called `Main`. The compiler will not consider this to be the same as the `main` method, because `Main` starts with an uppercase letter and the Java language is case sensitive. Upper- and lowercase letters are considered to be completely different from each other, and to the compiler `Main` is no better match for `main` than `rain`. The compiler will cheerfully compile your `Main` method, but when the Java virtual machine reads the compiled file, it will complain about the missing `main` method and refuse to run the program. Of course, the message “missing main method” should give you a clue where to look for the error.

If you get an error message that seems to indicate that the compiler or virtual machine is on the wrong track, check for spelling and capitalization. If you misspell the name of a symbol (for example, `ou` instead of `out`), the compiler will produce a message such as “cannot find symbol `ou`”. That error message is usually a good clue that you made a spelling error.

1.7 Problem Solving: Algorithm Design

You will soon learn how to program calculations and decision making in Java. But before we look at the mechanics of implementing computations in the next chapter, let's consider how you can describe the steps that are necessary for finding the solution to a problem.

1.7.1 The Algorithm Concept

You may have run across advertisements that encourage you to pay for a computerized service that matches you up with a love partner. Think how this might work. You fill out a form and send it in. Others do the same. The data are processed by a computer program. Is it reasonable to assume that the computer can perform the task of finding the best match for you? Suppose your younger brother, not the computer, had all the forms on his desk. What instructions could you give him? You can't say, "Find the best-looking person who likes inline skating and browsing the Internet". There is no objective standard for good looks, and your brother's opinion (or that of a computer program analyzing the photos of prospective partners) will likely be different from yours. If you can't give written instructions for someone to solve the problem, there is no way the computer can magically find the right solution. The computer can only do what you tell it to do. It just does it faster, without getting bored or exhausted.

For that reason, a computerized match-making service cannot guarantee to find the optimal match for you. Instead, you may be presented with a set of potential partners who share common interests with you. That is a task that a computer program can solve.

In order for a computer program to provide an answer to a problem that computes an answer, it must follow a sequence of steps that is

- Unambiguous
- Executable
- Terminating

An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.

The step sequence is *unambiguous* when there are precise instructions for what to do at each step and where to go next. There is no room for guesswork or personal opinion. A step is *executable* when it can be carried out in practice. For example, a computer can list all people that share your hobbies, but it can't predict who will be your life-long partner. Finally, a sequence of steps is *terminating* if it will eventually come to an end. A program that keeps working without delivering an answer is clearly not useful.

A sequence of steps that is unambiguous, executable, and terminating is called an **algorithm**. Although there is no algorithm for finding a partner, many problems do have algorithms for solving them. The next section gives an example.



© mammamaari/iStockphoto.

Finding the perfect partner is not a problem that a computer can solve.



© Claudiad/iStockphoto.

An algorithm is a recipe for finding a solution.

1.7.2 An Algorithm for Solving an Investment Problem

Consider the following investment problem:

You put \$10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original?

Could you solve this problem by hand? Sure, you could. You figure out the balance as follows:

year	interest	balance
0		10000
1	$10000.00 \times 0.05 = 500.00$	$10000.00 + 500.00 = 10500.00$
2	$10500.00 \times 0.05 = 525.00$	$10500.00 + 525.00 = 11025.00$
3	$11025.00 \times 0.05 = 551.25$	$11025.00 + 551.25 = 11576.25$
4	$11576.25 \times 0.05 = 578.81$	$11576.25 + 578.81 = 12155.06$

You keep going until the balance is at least \$20,000. Then the last number in the year column is the answer.

Of course, carrying out this computation is intensely boring to you or your younger brother. But computers are very good at carrying out repetitive calculations quickly and flawlessly. What is important to the computer is a description of the steps for finding the solution. Each step must be clear and unambiguous, requiring no guesswork. Here is such a description:

Set year to 0, balance to 10000.

year	interest	balance
0		10000

While the balance is less than \$20,000

Add 1 to the year.

Set the interest to balance $\times 0.05$ (i.e., 5 percent interest).

Add the interest to the balance.

year	interest	balance
0		10000
1	500.00	10500.00
14	942.82	19799.32
15	989.96	20789.28

Report year as the answer.

These steps are not yet in a language that a computer can understand, but you will soon learn how to formulate them in Java. This informal description is called **pseudocode**. We examine the rules for writing pseudocode in the next section.

1.7.3 Pseudocode

Pseudocode is an informal description of a sequence of steps for solving a problem.

There are no strict requirements for pseudocode because it is read by human readers, not a computer program. Here are the kinds of pseudocode statements and how we will use them in this book:

- Use statements such as the following to describe how a value is set or changed:

total cost = purchase price + operating cost
Multiply the balance value by 1.05.
Remove the first and last character from the word.

- Describe decisions and repetitions as follows:

If total cost 1 < total cost 2
While the balance is less than \$20,000
For each picture in the sequence

Use indentation to indicate which statements should be selected or repeated:

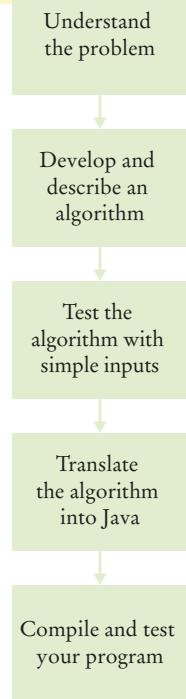
For each car
operating cost = 10 x annual fuel cost
total cost = purchase price + operating cost

Here, the indentation indicates that both statements should be executed for each car.

- Indicate results with statements such as:

Choose car1.
Report year as the answer.

1.7.4 From Algorithms to Programs



In Section 1.7.2, we developed pseudocode for finding how long it takes to double an investment. Let's double-check that the pseudocode represents an algorithm; that is, that it is unambiguous, executable, and terminating.

Our pseudocode is unambiguous. It simply tells how to update values in each step. The pseudocode is executable because we use a fixed interest rate. Had we said to use the actual interest rate that will be charged in years to come, and not a fixed rate of 5 percent per year, the instructions would not have been executable. There is no way for anyone to know what the interest rate will be in the future. It requires a bit of thought to see that the steps are terminating: With every step, the balance goes up by at least \$500, so eventually it must reach \$20,000.

Therefore, we have found an algorithm to solve our investment problem, and we know we can find the solution by programming a computer. The existence of an algorithm is an essential prerequisite for programming a task. You need to first discover and describe an algorithm for the task before you start programming (see Figure 8). In the chapters that follow, you will learn how to express algorithms in the Java language.

Figure 8 The Software Development Process

SELF CHECK

21. Suppose the interest rate was 20 percent. How long would it take for the investment to double?

22. Suppose your cell phone carrier charges you \$29.95 for up to 300 minutes of calls, and \$0.45 for each additional minute, plus 12.5 percent taxes and fees. Give an algorithm to compute the monthly charge from a given number of minutes.

23. Consider the following pseudocode for finding the most attractive photo from a sequence of photos:

Set "the best so far" to the first photo.

For each photo in the sequence

If the photo is more attractive than "the best so far"

Set "the best so far" to the photo.

Report "the best so far" as the most attractive photo in the sequence.

Is this an algorithm that will find the most attractive photo?

24. Suppose each photo in Self Check 23 had a price tag. Give an algorithm for finding the most expensive photo.

25. Suppose you have a random sequence of black and white marbles and want to rearrange it so that the black and white marbles are grouped together. Consider this algorithm:

Repeat until sorted

Locate the first black marble that is preceded by a white marble, and switch them.

What does the algorithm do with the sequence O●O●●? Spell out the steps until the algorithm stops.

26. Suppose you have a random sequence of colored marbles. Consider this pseudocode:

Repeat until sorted

Locate the first marble that is preceded by a marble of a different color, and switch them.

Why is this not an algorithm?

Practice It Now you can try these exercises at the end of the chapter: R1.16, E1.4, P1.1.

HOW TO 1.1**Describing an Algorithm with Pseudocode**

This is the first of many “How To” sections in this book that give you step-by-step procedures for carrying out important tasks in developing computer programs.

Before you are ready to write a program in Java, you need to develop an algorithm—a method for arriving at a solution for a particular problem. Describe the algorithm in pseudocode—a sequence of precise steps formulated in English. To illustrate, we’ll devise an algorithm for this problem:

Problem Statement You have the choice of buying one of two cars. One is more fuel efficient than the other, but also more expensive. You know the price and fuel efficiency (in miles per gallon, mpg) of both cars. You plan to keep the car for ten years. Assume a price of \$4 per gallon of gas and usage of 15,000 miles per year. You will pay cash for the car and not worry about financing costs. Which car is the better deal?



© dlewis33/iStockphoto.

Step 1 Determine the inputs and outputs.

In our sample problem, we have these inputs:

- **purchase price1** and **fuel efficiency1**, the price and fuel efficiency (in mpg) of the first car
- **purchase price2** and **fuel efficiency2**, the price and fuel efficiency of the second car

We simply want to know which car is the better buy. That is the desired output.

Step 2 Break down the problem into smaller tasks.

For each car, we need to know the total cost of driving it. Let's do this computation separately for each car. Once we have the total cost for each car, we can decide which car is the better deal.

The total cost for each car is **purchase price + operating cost**.

We assume a constant usage and gas price for ten years, so the operating cost depends on the cost of driving the car for one year.

The operating cost is **10 x annual fuel cost**.

The annual fuel cost is **price per gallon x annual fuel consumed**.

The annual fuel consumed is **annual miles driven / fuel efficiency**. For example, if you drive the car for 15,000 miles and the fuel efficiency is 15 miles/gallon, the car consumes 1,000 gallons.

Step 3 Describe each subtask in pseudocode.

In your description, arrange the steps so that any intermediate values are computed before they are needed in other computations. For example, list the step

total cost = purchase price + operating cost

after you have computed **operating cost**.

Here is the algorithm for deciding which car to buy:

For each car, compute the total cost as follows:

annual fuel consumed = annual miles driven / fuel efficiency

annual fuel cost = price per gallon x annual fuel consumed

operating cost = 10 x annual fuel cost

total cost = purchase price + operating cost

If total cost of car1 < total cost of car2

Choose car1.

Else

Choose car2.

Step 4 Test your pseudocode by working a problem.

We will use these sample values:

Car 1: \$25,000, 50 miles/gallon

Car 2: \$20,000, 30 miles/gallon

Here is the calculation for the cost of the first car:

annual fuel consumed = annual miles driven / fuel efficiency = 15000 / 50 = 300

annual fuel cost = price per gallon x annual fuel consumed = 4 x 300 = 1200

operating cost = 10 x annual fuel cost = 10 x 1200 = 12000

total cost = purchase price + operating cost = 25000 + 12000 = 37000

Similarly, the total cost for the second car is \$40,000. Therefore, the output of the algorithm is to choose car 1.

The following Worked Example demonstrates how to use the concepts in this chapter and the steps in the How To to solve another problem. In this case, you will see how to develop an algorithm for laying tile in an alternating pattern of colors. You should read the Worked Example to review what you have learned, and for help in tackling another problem.

In future chapters, Worked Examples are provided for you on the book's companion Web site. A brief description of the problem tackled in the example will appear with a reminder to download it from www.wiley.com/go/bjlo2examples. You will find any code related to the Worked Example included with the book's companion code for the chapter. When you see the Worked Example description, download the example and the code to learn how the problem was solved.

WORKED EXAMPLE 1.1

Writing an Algorithm for Tiling a Floor



Problem Statement Write an algorithm for tiling a rectangular bathroom floor with alternating black and white tiles measuring 4×4 inches. The floor dimensions, measured in inches, are multiples of 4.

Step 1

Determine the inputs and outputs.

The inputs are the floor dimensions ($\text{length} \times \text{width}$), measured in inches. The output is a tiled floor.

Step 2

Break down the problem into smaller tasks.

A natural subtask is to lay one row of tiles. If you can solve that task, then you can solve the problem by laying one row next to the other, starting from a wall, until you reach the opposite wall.

How do you lay a row? Start with a tile at one wall. If it is white, put a black one next to it. If it is black, put a white one next to it. Keep going until you reach the opposite wall. The row will contain $\text{width} / 4$ tiles.



© iStockphoto.com/riban

Step 3

Describe each subtask in pseudocode.

In the pseudocode, you want to be more precise about exactly where the tiles are placed.

Place a black tile in the northwest corner.

While the floor is not yet filled

Repeat $\text{width} / 4 - 1$ times

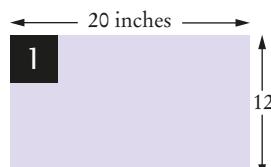
Place a tile east of the previously placed tile. If the previously placed tile was white, pick a black one; otherwise, a white one.

Locate the tile at the beginning of the row that you just placed. If there is space to the south, place a tile of the opposite color below it.

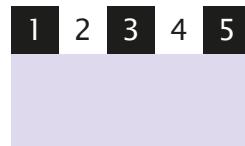
Step 4

Test your pseudocode by working a problem.

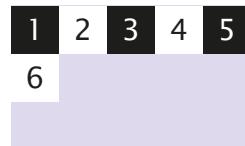
Suppose you want to tile an area measuring 20×12 inches. The first step is to place a black tile in the northwest corner.



Next, alternate four tiles until reaching the east wall. ($\text{width} / 4 - 1 = 20 / 4 - 1 = 4$)



There is room to the south. Locate the tile at the beginning of the completed row. It is black. Place a white tile south of it.



Complete the row.



There is still room to the south. Locate the tile at the beginning of the completed row. It is white. Place a black tile south of it.



Complete the row.



Now the entire floor is filled, and you are done.



VIDEO EXAMPLE 1.2

Dividing Household Expenses

See how to develop an algorithm for dividing household expenses among roommates. Go to wiley.com/go/bjlo2videos to view Video Example 1.2.

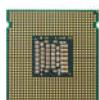


© Yin Yang/
iStockphoto.

CHAPTER SUMMARY

Define “computer program” and programming.

- Computers execute very basic instructions in rapid succession.
- A computer program is a sequence of instructions and decisions.
- Programming is the act of designing and implementing computer programs.

Describe the components of a computer.

- The central processing unit (CPU) performs program control and data processing.
- Storage devices include memory and secondary storage.

Describe the process of translating high-level languages to machine code.

- Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.
- Java was designed to be safe and portable, benefiting both Internet users and students.
- Java programs are distributed as instructions for a virtual machine, making them platform-independent.
- Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.

Become familiar with your Java programming environment.

- Set aside time to become familiar with the programming environment that you will use for your class work.
- An editor is a program for entering and modifying text, such as a Java program.
- Java is case sensitive. You must be careful about distinguishing between uppercase and lowercase letters.
- The Java compiler translates source code into class files that contain instructions for the Java virtual machine.
- Develop a strategy for keeping backup copies of your work before disaster strikes.

Describe the building blocks of a simple program.

- Classes are the fundamental building blocks of Java programs.
- Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.
- Each class contains declarations of methods. Each method contains a sequence of instructions.
- A method is called by specifying the method and its arguments.
- A string is a sequence of characters enclosed in quotation marks.

Classify program errors as compile-time and run-time errors.

- A compile-time error is a violation of the programming language rules that is detected by the compiler.
- A run-time error causes a program to take an action that the programmer did not intend.

Write pseudocode for simple algorithms.

- An algorithm for solving a problem is a sequence of steps that is unambiguous, executable, and terminating.
- Pseudocode is an informal description of a sequence of steps for solving a problem.



STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.io.PrintStream
    print
    println
```

```
java.lang.System
    out
```

REVIEW EXERCISES

- R1.1 Explain the difference between using a computer program and programming a computer.
- R1.2 Which parts of a computer can store program code? Which can store user data?
- R1.3 Which parts of a computer serve to give information to the user? Which parts take user input?
- R1.4 A toaster is a single-function device, but a computer can be programmed to carry out different tasks. Is your cell phone a single-function device, or is it a programmable computer? (Your answer will depend on your cell phone model.)
- R1.5 Explain two benefits of using Java over machine code.
- R1.6 On your own computer or on a lab computer, find the exact location (folder or directory name) of
 - a. The sample file `HelloPrinter.java`, which you wrote with the editor.
 - b. The Java program launcher `java.exe` or `java`.
 - c. The library file `rt.jar` that contains the run-time library.
- R1.7 What does this program print?

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("39 + 3");
        System.out.println(39 + 3);
    }
}
```

- R1.8 What does this program print? Pay close attention to spaces.

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

- R1.9 What is the compile-time error in this program?

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Hello", "World!");
    }
}
```

- **R1.10** Write three versions of the `HelloPrinter.java` program that have different compile-time errors. Write a version that has a run-time error.
- **R1.11** How do you discover syntax errors? How do you discover logic errors?
- **R1.12** The cafeteria offers a discount card for sale that entitles you, during a certain period, to a free meal whenever you have bought a given number of meals at the regular price. The exact details of the offer change from time to time. Describe an algorithm that lets you determine whether a particular offer is a good buy. What other inputs do you need?
- **R1.13** Write an algorithm to settle the following question: A bank account starts out with \$10,000. Interest is compounded monthly at 6 percent per year (0.5 percent per month). Every month, \$500 is withdrawn to meet college expenses. After how many years is the account depleted?
- **R1.14** Consider the question in Exercise R1.13. Suppose the numbers (\$10,000, 6 percent, \$500) were user selectable. Are there values for which the algorithm you developed would not terminate? If so, change the algorithm to make sure it always terminates.
- **R1.15** In order to estimate the cost of painting a house, a painter needs to know the surface area of the exterior. Develop an algorithm for computing that value. Your inputs are the width, length, and height of the house, the number of windows and doors, and their dimensions. (Assume the windows and doors have a uniform size.)
- **R1.16** In How To 1.1, you made assumptions about the price of gas and annual usage to compare cars. Ideally, you would like to know which car is the better deal without making these assumptions. Why can't a computer program solve that problem?
- **R1.17** Suppose you put your younger brother in charge of backing up your work. Write a set of detailed instructions for carrying out his task. Explain how often he should do it, and what files he needs to copy from which folder to which location. Explain how he should verify that the backup was carried out correctly.
- **R1.18** Write pseudocode for an algorithm that describes how to prepare Sunday breakfast in your household.
- **R1.19** The ancient Babylonians had an algorithm for determining the square root of a number a . Start with an initial guess of $a / 2$. Then find the average of your guess g and a / g . That's your next guess. Repeat until two consecutive guesses are close enough. Write pseudocode for this algorithm.

PRACTICE EXERCISES

- **E1.1** Write a program that prints a greeting of your choice, perhaps in a language other than English.
- **E1.2** Write a program that prints the sum of the first ten positive integers, $1 + 2 + \dots + 10$.
- **E1.3** Write a program that prints the product of the first ten positive integers, $1 \times 2 \times \dots \times 10$. (Use `*` to indicate multiplication in Java.)
- **E1.4** Write a program that prints the balance of an account after the first, second, and third year. The account has an initial balance of \$1,000 and earns 5 percent interest per year.
- **E1.5** Write a program that displays your name inside a box on the screen, like this: Dave
Do your best to approximate lines with characters such as `| - +`.

- E1.6 Write a program that prints your name in large letters, such as

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

- E1.7 Write a program that prints your name in Morse code, like this:

```
.... - .- .-. -.-
```

Use a separate call to `System.out.print` for each letter.

- E1.8 Write a program that prints a face similar to (but different from) the following:

```
/////
+"""""+
(| o o |)
| ^ |
| '-' |
+-----+
```

- E1.9 Write a program that prints an imitation of a Piet Mondrian painting. (Search the Internet if you are not familiar with his paintings.) Use character sequences such as `@@@` or `:::` to indicate different colors, and use `-` and `|` to form lines.

- E1.10 Write a program that prints a house that looks exactly like the following:

```
+  
+ +  
+ +  
+---+  
| .. |  
| | |  
+-+-+-
```

- E1.11 Write a program that prints an animal speaking a greeting, similar to (but different from) the following:

```
/\ / \     ----
( ' ' ) / Hello \
( - ) < Junior |
| | | \ Coder! /
( __ ) ----
```

- E1.12 Write a program that prints three items, such as the names of your three best friends or favorite movies, on three separate lines.

- E1.13 Write a program that prints a poem of your choice. If you don't have a favorite poem, search the Internet for "Emily Dickinson" or "e e cummings".

- E1.14 Write a program that prints the United States flag, using `*` and `=` characters.

- E1.15 Type in and run the following program. Then modify it to show the message "Hello, *your name!*".

```
import javax.swing.JOptionPane;

public class DialogViewer
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "Hello, World!");
    }
}
```

```

    }
}
```

- E1.16** Type in and run the following program. Then modify it to print “Hello, *name!*”, displaying the name that the user typed in.

```

import javax.swing.JOptionPane;

public class DialogViewer
{
    public static void main(String[] args)
    {
        String name = JOptionPane.showInputDialog("What is your name?");
        System.out.println(name);
    }
}
```

- E1.17** Modify the program from Exercise E1.16 so that the dialog continues with the message “My name is Hal! What would you like me to do?” Discard the user’s input and display a message such as

I'm sorry, Dave. I'm afraid I can't do that.

Replace Dave with the name that was provided by the user.

- E1.18** Type in and run the following program. Then modify it to show a different greeting and image.

```

import java.net.URL;
import javax.swing.ImageIcon;
import javax.swing.JOptionPane;

public class Test
{
    public static void main(String[] args) throws Exception
    {
        URL imageLocation = new URL(
            "http://horstmann.com/java4everyone/duke.gif");
        JOptionPane.showMessageDialog(null, "Hello", "Title",
            JOptionPane.PLAIN_MESSAGE, new ImageIcon(imageLocation));
    }
}
```

- Business E1.19** Write a program that prints a two-column list of your friends’ birthdays. In the first column, print the names of your best friends; in the second, print their birthdays.

- Business E1.20** In the United States there is no federal sales tax, so every state may impose its own sales taxes. Look on the Internet for the sales tax charged in five U.S. states, then write a program that prints the tax rate for five states of your choice.

- Business E1.21** To speak more than one language is a valuable skill in the labor market today. One of the basic skills is learning to greet people. Write a program that prints a two-column list with the greeting phrases shown in the table. In the first column, print the phrase in English, in the second column, print the phrase in a language of your choice. If you don’t speak a language other than English, use an online translator or ask a friend.

Sales Tax Rates	
-----	-----
Alaska:	0%
Hawaii:	4%
.	.

List of Phrases to Translate

Good morning.

It is a pleasure to meet you.

Please call me tomorrow.

Have a nice day!

PROGRAMMING PROJECTS

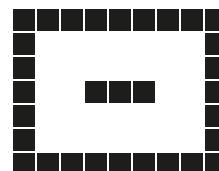
- P1.1** You want to decide whether you should drive your car to work or take the train. You know the one-way distance from your home to your place of work, and the fuel efficiency of your car (in miles per gallon). You also know the one-way price of a train ticket. You assume the cost of gas at \$4 per gallon, and car maintenance at 5 cents per mile. Write an algorithm to decide which commute is cheaper.
- P1.2** You want to find out which fraction of your car's use is for commuting to work, and which is for personal use. You know the one-way distance from your home to work. For a particular period, you recorded the beginning and ending mileage on the odometer and the number of work days. Write an algorithm to settle this question.
- P1.3** The value of π can be computed according to the following formula:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Write an algorithm to compute π . Because the formula is an infinite series and an algorithm must stop after a finite number of steps, you should stop when you have the result determined to six significant digits.

- Business P1.4** Imagine that you and a number of friends go to a luxury restaurant, and when you ask for the bill you want to split the amount and the tip (15 percent) between all. Write pseudocode for calculating the amount of money that everyone has to pay. Your program should print the amount of the bill, the tip, the total cost, and the amount each person has to pay. It should also print how much of what each person pays is for the bill and for the tip.

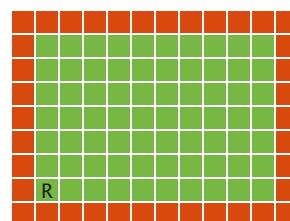
- P1.5** Write an algorithm to create a tile pattern composed of black and white tiles, with a fringe of black tiles all around and two or three black tiles in the center, equally spaced from the boundary. The inputs to your algorithm are the total number of rows and columns in the pattern.



- P1.6** Write an algorithm that allows a robot to mow a rectangular lawn, provided it has been placed in a corner, like this:

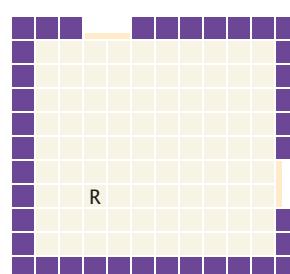
The robot (marked as R) can:

- Move forward by one unit.
- Turn left or right.
- Sense the color of the ground one unit in front of it.



- P1.7** Consider a robot that is placed in a room. The robot can:

- Move forward by one unit.
- Turn left or right.
- Sense what is in front of it: a wall, a window, or neither.



Write an algorithm that enables the robot, placed anywhere in the room, to count the number of windows. For example, in the room at right, the robot (marked as R) should find that it has two windows.

© Skip O'Donnell/iStockphoto.



- P1.8** Consider a robot that has been placed in a maze. The right-hand rule tells you how to escape from a maze: Always have the right hand next to a wall, and eventually you will find an exit. The robot can:

- Move forward by one unit.
- Turn left or right.
- Sense what is in front of it: a wall, an exit, or neither.

Write an algorithm that lets the robot escape the maze. You may assume that there is an exit that is reachable by the right-hand rule. Your challenge is to deal with situations in which the path turns. The robot can't see turns. It can only see what is directly in front of it.

- Business P1.9** Suppose you received a loyalty promotion that lets you purchase one item, valued up to \$100, from an online catalog. You want to make the best of the offer. You have a list of all items for sale, some of which are less than \$100, some more. Write an algorithm to produce the item that is closest to \$100. If there is more than one such item, list them all. Remember that a computer will inspect one item at a time—it can't just glance at a list and find the best one.

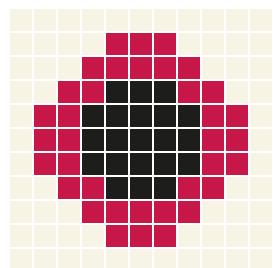
- Science P1.10** A television manufacturer advertises that a television set has a certain size, measured diagonally. You wonder how the set will fit into your living room. Write an algorithm that yields the horizontal and vertical size of the television. Your inputs are the diagonal size and the aspect ratio (the ratio of width to height, usually 16 : 9 for television sets).



© Don Bayley/iStockphoto.

- Science P1.11** Cameras today can correct “red eye” problems caused when the photo flash makes eyes look red. Write pseudocode for an algorithm that can detect red eyes. Your input is a pattern of colors, such as that at right.

You are given the number of rows and columns. For any row or column number, you can query the color, which will be red, black, or something else. If you find that the center of the black pixels coincides with the center of the red pixels, you have found a red eye, and your output should be “yes”. Otherwise, your output is “no”.



ANSWERS TO SELF-CHECK QUESTIONS

1. A program that reads the data on the CD and sends output to the speakers and the screen.
2. A CD player can do one thing—play music CDs. It cannot execute programs.
3. Nothing.
4. In secondary storage, typically a hard disk.
5. The central processing unit.
6. A smartphone has a CPU and memory, like any computer. A few smartphones have keyboards. Generally, the touchpad is used instead of a mouse. Secondary storage is in the form of a solid state drive. Of course, smartphones have a display, speaker, and microphone. The network connection uses the wireless radio to connect to a cell tower.

7. Safety and portability.
8. No one person can learn the entire library—it is too large.
9. The answer varies among systems. A typical answer might be `/home/dave/cs1/hello>Hello-Printer.java` or `c:\Users\Dave\Workspace\hello\HelloPrinter.java`
10. You back up your files and folders.
11. Change `World` to your name (here, `Dave`):

```
System.out.println("Hello, Dave!");
```
12.

```
System.out.println("H");
System.out.println("e");
System.out.println("l");
System.out.println("l");
System.out.println("o");
```
13. No. The compiler would look for an item whose name is `Hello`. You need to enclose `Hello` in quotation marks:

```
System.out.println("Hello");
```
14. The printout is `My lucky number is12`. It would be a good idea to add a space after the `is`.
15. `Hello`
a blank line
`World`
16. This is a compile-time error. The compiler will complain that it does not know the meanings of the words `Hello` and `World`.
17. This is a compile-time error. The compiler will complain that `System.out` does not have a method called `printline`.
18. This is a run-time error. It is perfectly legal to give the name `hello` to a method, so the compiler won't complain. But when the program is run, the virtual machine will look for a `main` method and won't find one.
19. It is a run-time error. After all, the program had been compiled in order for you to run it.
20. When a program has compiler errors, no class file is produced, and there is nothing to run.
21. 4 years:
 - 0 10,000
 - 1 12,000
 - 2 14,400
 - 3 17,280
 - 4 20,736

22. Is the number of minutes at most 300?
 - a. If so, the answer is $\$29.95 \times 1.125 = \33.70 .
 - b. If not,
 1. Compute the difference: (number of minutes) – 300.
 2. Multiply that difference by 0.45.
 3. Add \$29.95.
 4. Multiply the total by 1.125. That is the answer.
23. No. The step **If the photo is more attractive than "the best so far"** is not executable because there is no objective way of deciding which of two photos is more attractive.
24. Set "the most expensive so far" to the first photo.
For each photo in the sequence

If the photo is more expensive than "the most expensive so far"

Set "the most expensive so far" to the photo.

Report "the most expensive so far" as the most expensive photo in the sequence.
25. The first black marble that is preceded by a white one is marked in blue:


Switching the two yields

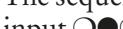

The next black marble to be switched is


yielding


The next steps are






Now the sequence is sorted.
26. The sequence doesn't terminate. Consider the input . The first two marbles keep getting switched.

FUNDAMENTAL DATA TYPES

CHAPTER GOALS

- To declare and initialize variables and constants
- To understand the properties and limitations of integers and floating-point numbers
- To appreciate the importance of comments and good code layout
- To write arithmetic expressions and assignment statements
- To create programs that read and process inputs, and display the results
- To learn how to use the Java String type

CHAPTER CONTENTS

2.1 VARIABLES 32

- SYN** Variable Declaration 33
- SYN** Assignment 36
- SYN** Constant Declaration 37
- CE1** Using Undeclared or Uninitialized Variables 39
- PT1** Choose Descriptive Variable Names 40
- CE2** Overflow 40
- CE3** Roundoff Errors 40
- PT2** Do Not Use Magic Numbers 41
- ST1** Numeric Types in Java 41
- ST2** Big Numbers 42

2.2 ARITHMETIC 43

- SYN** Cast 46
- CE4** Unintended Integer Division 48
- CE5** Unbalanced Parentheses 48
- PT3** Spaces in Expressions 49
- J81** Avoiding Negative Remainders 49
- ST3** Combining Assignment and Arithmetic 49
- VE1** Using Integer Division 
- C&S** The Pentium Floating-Point Bug 50

Flight Number	Destination	Status
549	Osaka/Kansai	Final Call
11	Taipei	Final Call
82	Manila	Final Call
683	Toronto	502 Final Call
250	Nanjing	2 Final Call
852	Bangkok/D	17 Final Call
165	Harbin	62 Final Call
904	Kuala Lumpur	Cancelled
21	Jinjiang	21 Boarding
820	Nanjing	503
677	Kaohsiung	506 Gate Change
206	Singapore	49 Boarding
683	Shanghai/P	40 Boarding
69	S	1

©sanxmeg/iStockphoto.

2.3 INPUT AND OUTPUT 50

- SYN** Input Statement 51
- PT4** Use the API Documentation 55
- HT1** Carrying out Computations 56
- WE1** Computing the Cost of Stamps 

2.4 PROBLEM SOLVING: FIRST DO IT BY HAND 59

- WE2** Computing Travel Time 

2.5 STRINGS 61

- ST4** Instance Methods and Static Methods 66
- ST5** Using Dialog Boxes for Input and Output 67
- VE2** Computing Distances on Earth 
- C&S** International Alphabets and Unicode 68



©sanxmeg/iStockphoto.

Numbers and character strings (such as the ones on this display board) are important data types in any Java program. In this chapter, you will learn how to work with numbers and text, and how to write simple programs that perform useful tasks with them.

2.1 Variables

When your program carries out computations, you will want to store values so that you can use them later. In a Java program, you use **variables** to store values. In this section, you will learn how to declare and use variables.

To illustrate the use of variables, we will develop a program that solves the following problem. Soft drinks are sold in cans and bottles. A store offers a six-pack of 12-ounce cans for the same price as a two-liter bottle. Which should you buy? (Twelve fluid ounces equal approximately 0.355 liters.)

In our program, we will declare variables for the number of cans per pack and for the volume of each can. Then we will compute the volume of a six-pack in liters and print out the answer.



What contains more soda? A six-pack of 12-ounce cans or a two-liter bottle?

(cans) © blackred/iStockphoto;
(bottle) © travis manley/iStockphoto.

2.1.1 Variable Declarations

The following statement declares a variable named `cansPerPack`:

```
int cansPerPack = 6;
```

A variable is a storage location in a computer program. Each variable has a name and holds a value.

A **variable** is similar to a parking space in a parking garage. The parking space has an identifier (such as “J 053”), and it can hold a vehicle. A variable has a name (such as `cansPerPack`), and it can hold a value (such as 6).

Like a variable in a computer program, a parking space has an identifier and a contents.



Javier Larrea/Age Fotostock.

Syntax 2.1 Variable Declaration

Syntax

```
typeName variableName = value;
or
typeName variableName;
```

Types introduced in this chapter are the number types **int** and **double** (see Table 2) and the String type (see Section 2.5).

 Use a descriptive variable name. See Programming Tip 2.1.

See Table 3 for rules and examples of valid names.

`int cansPerPack = 6;`

A variable declaration ends with a semicolon.

 Supplying an initial value is optional, but it is usually a good idea. See Common Error 2.1.

When declaring a variable, you usually specify an initial value.

When declaring a variable, you also specify the type of its values.

When declaring a variable, you usually want to **initialize** it. That is, you specify the value that should be stored in the variable. Consider again this variable declaration:

```
int cansPerPack = 6;
```

The variable `cansPerPack` is initialized with the value 6.

Like a parking space that is restricted to a certain type of vehicle (such as a compact car, motorcycle, or electric vehicle), a variable in Java stores data of a specific **type**. Java supports quite a few data types: numbers, text strings, files, dates, and many others. You must specify the type whenever you declare a variable (see Syntax 2.1).

The `cansPerPack` variable is an **integer**, a whole number without a fractional part. In Java, this type is called `int`. (See the next section for more information about number types in Java.)

Note that the type comes before the variable name:

```
int cansPerPack = 6;
```

After you have declared and initialized a variable, you can use it. For example,

```
int cansPerPack = 6;
System.out.println(cansPerPack);
int cansPerCrate = 4 * cansPerPack;
```

Table 1 shows several examples of variable declarations.



Each parking space is suitable for a particular type of vehicle, just as each variable holds a value of a particular type.

Table 1 Variable Declarations in Java

Variable Name	Comment
<code>int cans = 6;</code>	Declares an integer variable and initializes it with 6.
<code>int total = cans + bottles;</code>	The initial value need not be a fixed value. (Of course, cans and bottles must have been previously declared.)
 <code>bottles = 1;</code>	Error: The type is missing. This statement is not a declaration but an assignment of a new value to an existing variable—see Section 2.1.4.
 <code>int volume = "2";</code>	Error: You cannot initialize a number with a string.
<code>int cansPerPack;</code>	Declares an integer variable without initializing it. This can be a cause for errors—see Common Error 2.1 on page 39.
<code>int dollars, cents;</code>	Declares two integer variables in a single statement. In this book, we will declare each variable in a separate statement.

2.1.2 Number Types

Use the `int` type for numbers that cannot have a fractional part.

In Java, there are several different types of numbers. You use the `int` type to denote a whole number without a fractional part. For example, there must be an integer number of cans in any pack of cans—you cannot have a fraction of a can.

When a fractional part is required (such as in the number 0.335), we use **floating-point numbers**. The most commonly used type for floating-point numbers in Java is called `double`. (If you want to know the reason, read Special Topic 2.1 on page 41.) Here is the declaration of a floating-point variable:

```
double canVolume = 0.335;
```

Table 2 Number Literals in Java

Number	Type	Comment
6	<code>int</code>	An integer has no fractional part.
-6	<code>int</code>	Integers can be negative.
0	<code>int</code>	Zero is an integer.
0.5	<code>double</code>	A number with a fractional part has type <code>double</code> .
1.0	<code>double</code>	An integer with a fractional part .0 has type <code>double</code> .
1E6	<code>double</code>	A number in exponential notation: 1×10^6 or 1000000. Numbers in exponential notation always have type <code>double</code> .
2.96E-2	<code>double</code>	Negative exponent: $2.96 \times 10^{-2} = 2.96 / 100 = 0.0296$
 100,000		Error: Do not use a comma as a decimal separator.
 3 1/2		Error: Do not use fractions; use decimal notation: 3.5

Use the `double` type for floating-point numbers.

When a value such as 6 or 0.335 occurs in a Java program, it is called a **number literal**. If a number literal has a decimal point, it is a floating-point number; otherwise, it is an integer. Table 2 shows how to write integer and floating-point literals in Java.

2.1.3 Variable Names

When you declare a variable, you should pick a name that explains its purpose. For example, it is better to use a descriptive name, such as `canVolume`, than a terse name, such as `cv`.

In Java, there are a few simple rules for variable names:

1. Variable names must start with a letter or the underscore (`_`) character, and the remaining characters must be letters, numbers, or underscores. (Technically, the `$` symbol is allowed as well, but you should not use it—it is intended for names that are automatically generated by tools.)
2. You cannot use other symbols such as `?` or `%`. Spaces are not permitted inside names either. You can use uppercase letters to denote word boundaries, as in `cansPerPack`. This naming convention is called *camel case* because the uppercase letters in the middle of the name look like the humps of a camel.)
3. Variable names are **case sensitive**, that is, `canVolume` and `canvolume` are different names.
4. You cannot use **reserved words** such as `double` or `class` as names; these words are reserved exclusively for their special Java meanings. (See Appendix C for a listing of all reserved words in Java.)



© GlobalP/iStockphoto.

By convention, variable names should start with a lowercase letter.

It is a convention among Java programmers that variable names should start with a lowercase letter (such as `canVolume`) and class names should start with an uppercase letter (such as `HelloPrinter`). That way, it is easy to tell them apart.

Table 3 shows examples of legal and illegal variable names in Java.

Table 3 Variable Names in Java

Variable Name	Comment
<code>canVolume1</code>	Variable names consist of letters, numbers, and the underscore character.
<code>x</code>	In mathematics, you use short variable names such as <code>x</code> or <code>y</code> . This is legal in Java, but not very common, because it can make programs harder to understand (see Programming Tip 2.1 on page 40).
 <code>CanVolume</code>	Caution: Variable names are case sensitive. This variable name is different from <code>canVolume</code> , and it violates the convention that variable names should start with a lowercase letter.
 <code>6pack</code>	Error: Variable names cannot start with a number.
 <code>can volume</code>	Error: Variable names cannot contain spaces.
 <code>double</code>	Error: You cannot use a reserved word as a variable name.
 <code>ltr/fl.oz</code>	Error: You cannot use symbols such as <code>/</code> or <code>.</code>

2.1.4 The Assignment Statement

An assignment statement stores a new value in a variable, replacing the previously stored value.

You use the **assignment statement** to place a new value into a variable. Here is an example:

```
cansPerPack = 8;
```

The left-hand side of an assignment statement consists of a variable. The right-hand side is an expression that has a value. That value is stored in the variable, overwriting its previous contents.

There is an important difference between a variable declaration and an assignment statement:

```
int cansPerPack = 6;  
...  
cansPerPack = 8;
```

Variable declaration

Assignment statement

The first statement is the declaration of cansPerPack. It is an instruction to create a new variable of type int, to give it the name cansPerPack, and to initialize it with 6. The second statement is an *assignment statement*: an instruction to replace the contents of the *existing* variable cansPerPack with another value.

The = sign doesn't mean that the left-hand side is *equal* to the right-hand side. The expression on the right is evaluated, and its value is placed into the variable on the left.

Do not confuse this *assignment operation* with the = used in algebra to denote *equality*. The assignment operator is an instruction to do something—namely, place a value into a variable. The mathematical equality states that two values are equal.

For example, in Java, it is perfectly legal to write

```
totalVolume = totalVolume + 2;
```

It means to look up the value stored in the variable totalVolume, add 2 to it, and place the result back into totalVolume. (See Figure 1.) The net effect of executing this statement is to increment totalVolume by 2. For example, if totalVolume was 2.13 before execution of the statement, it is set to 4.13 afterwards. Of course, in mathematics it would make no sense to write that $x = x + 2$. No value can equal itself plus 2.

Syntax 2.2 Assignment

Syntax `variableName = value;`

This is an initialization
of a new variable,
NOT an assignment.

The name of a previously
defined variable

```
double total = 0;
```

```
.
```

```
total = bottles * BOTTLE_VOLUME;
```

```
.
```

```
.
```

```
total = total + cans * CAN_VOLUME;
```

This is an assignment.

The expression that replaces the previous value

The same name
can occur on both sides.
See Figure 1.

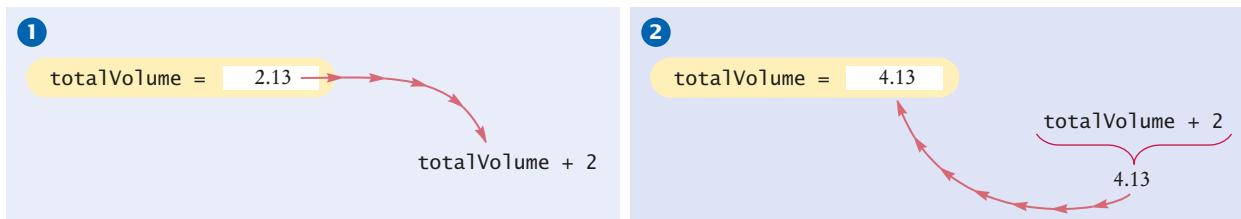


Figure 1 Executing the Assignment `totalVolume = totalVolume + 2`

2.1.5 Constants

You cannot change the value of a variable that is defined as `final`.

When a variable is defined with the reserved word `final`, its value can never change. Constants are commonly written using capital letters to distinguish them visually from regular variables:

```
final double BOTTLE_VOLUME = 2;
```

It is good programming style to use named constants in your program to explain the meanings of numeric values. For example, compare the statements

```
double totalVolume = bottles * 2;
```

and

```
double totalVolume = bottles * BOTTLE_VOLUME;
```

A programmer reading the first statement may not understand the significance of the number 2. The second statement, with a named constant, makes the computation much clearer.

Syntax 2.3 Constant Declaration

Syntax `final typeName variableName = expression;`

The `final` reserved word indicates that this value cannot be modified.

`final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can`

Use uppercase letters for constants.

This comment explains how the value for the constant was determined.

2.1.6 Comments

As your programs get more complex, you should add **comments**, explanations for human readers of your code. For example, here is a comment that explains the value used in a variable initialization:

```
final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
```

This comment explains the significance of the value 0.355 to a human reader. The compiler does not process comments at all. It ignores everything from a `//` delimiter to the end of the line.

Use comments to add explanations for humans who read your code. The compiler ignores comments.

It is a good practice to provide comments. This helps programmers who read your code understand your intent. In addition, you will find comments helpful when you review your own programs.

You use the `//` delimiter for short comments. If you have a longer comment, enclose it between `/*` and `*/` delimiters. The compiler ignores these delimiters and everything in between. For example,

```
/*
    There are approximately 0.335 liters in a 12-ounce can because one ounce
    equals 0.02957353 liter; see The International Systems of Units (SI) - Conversion
    Factors for General Use (NIST Special Publication 1038).
*/
```

Finally, start a comment that explains the purpose of a program with the `/**` delimiter instead of `/*`. Tools that analyze source files rely on that convention. For example,

```
/**
    This program computes the volume (in liters) of a six-pack of soda cans.
*/
```

The following program shows the use of variables, constants, and the assignment statement. The program displays the volume of a six-pack of cans and the total volume of the six-pack and a two-liter bottle. We use constants for the can and bottle volumes. The `totalVolume` variable is initialized with the volume of the cans. Using an assignment statement, we add the bottle volume. As you can see from the program output, the six-pack of cans contains over two liters of soda.

sec01/Volume1.java

```
1  /**
2   * This program computes the volume (in liters) of a six-pack of soda
3   * cans and the total volume of a six-pack and a two-liter bottle.
4  */
5  public class Volume1
6  {
7      public static void main(String[] args)
8      {
9          int cansPerPack = 6;
10         final double CAN_VOLUME = 0.355; // Liters in a 12-ounce can
11         double totalVolume = cansPerPack * CAN_VOLUME;
12
13         System.out.print("A six-pack of 12-ounce cans contains ");
14         System.out.print(totalVolume);
15         System.out.println(" liters.");
16
17         final double BOTTLE_VOLUME = 2; // Two-liter bottle
18
19         totalVolume = totalVolume + BOTTLE_VOLUME;
20
21         System.out.print("A six-pack and a two-liter bottle contain ");
22         System.out.print(totalVolume);
23         System.out.println(" liters.");
24     }
25 }
```

Program Run

```
A six-pack of 12-ounce cans contains 2.13 liters.
A six-pack and a two-liter bottle contain 4.13 liters.
```

Just as a television commentator explains the news, you use comments in your program to explain its behavior.



© iStockphoto.com

SELF CHECK



1. Declare a variable suitable for holding the number of bottles in a case.
2. What is wrong with the following variable declaration?
`int ounces per liter = 28.35`
3. Declare and initialize two variables, `unitPrice` and `quantity`, to contain the unit price of a single bottle and the number of bottles purchased. Use reasonable initial values.
4. Use the variables declared in Self Check 3 to display the total purchase price.
5. Some drinks are sold in four-packs instead of six-packs. How would you change the `Volume1.java` program to compute the total volume?
6. What is wrong with this comment?
`double canVolume = 0.355; /* Liters in a 12-ounce can //`
7. Suppose the type of the `cansPerPack` variable in `Volume1.java` was changed from `int` to `double`. What would be the effect on the program?
8. Why can't the variable `totalVolume` in the `Volume1.java` program be declared as `final`?
9. How would you explain assignment using the parking space analogy?

Practice It

Now you can try these exercises at the end of the chapter: R2.2, R2.3, E2.1.

Common Error 2.1



Using Undeclared or Uninitialized Variables

You must declare a variable before you use it for the first time. For example, the following sequence of statements would not be legal:

```
double canVolume = 12 * literPerOunce; // ERROR: literPerOunce is not yet declared
double literPerOunce = 0.0296;
```

In your program, the statements are compiled in order. When the compiler reaches the first statement, it does not know that `literPerOunce` will be declared in the next line, and it reports an error. The remedy is to reorder the declarations so that each variable is declared before it is used.

A related error is to leave a variable uninitialized:

```
int bottles;
int bottleVolume = bottles * 2; // ERROR: bottles is not yet initialized
```

The Java compiler will complain that you are using a variable that has not yet been given a value. The remedy is to assign a value to the variable before it is used.

Programming Tip 2.1



Choose Descriptive Variable Names

We could have saved ourselves a lot of typing by using shorter variable names, as in

```
double cv = 0.355;
```

Compare this declaration with the one that we actually used, though. Which one is easier to read? There is no comparison. Just reading `canVolume` is a lot less trouble than reading `cv` and then *figuring out* it must mean “can volume”.

In practical programming, this is particularly important when programs are written by more than one person. It may be obvious to *you* that `cv` stands for can volume and not current velocity, but will it be obvious to the person who needs to update your code years later? For that matter, will you remember yourself what `cv` means when you look at the code three months from now?

Common Error 2.2



Overflow

Because numbers are represented in the computer with a limited number of digits, they cannot represent arbitrary numbers.

The `int` type has a *limited range*: It can represent numbers up to a little more than two billion. For many applications, this is not a problem, but you cannot use an `int` to represent the world population.

If a computation yields a value that is outside the `int` range, the result *overflows*. No error is displayed. Instead, the result is truncated, yielding a useless value. For example,

```
int fiftyMillion = 50000000;
System.out.println(100 * fiftyMillion); // Expected: 5000000000
```

displays 705032704.

In situations such as this, you can switch to `double` values. However, read Common Error 2.3 for more information about a related issue: roundoff errors.

Common Error 2.3



Roundoff Errors

Roundoff errors are a fact of life when calculating with floating-point numbers. You probably have encountered that phenomenon yourself with manual calculations. If you calculate $1/3$ to two decimal places, you get 0.33. Multiplying again by 3, you obtain 0.99, not 1.00.

In the processor hardware, numbers are represented in the binary number system, using only digits 0 and 1. As with decimal numbers, you can get roundoff errors when binary digits are lost. They just may crop up at different places than you might expect.

Here is an example:

```
double price = 4.35;
double quantity = 100;
double total = price * quantity; // Should be 100 * 4.35 = 435
System.out.println(total); // Prints 434.9999999999999
```

In the binary system, there is no exact representation for 4.35, just as there is no exact representation for 1/3 in the decimal system. The representation used by the computer is just a little less than 4.35, so 100 times that value is just a little less than 435.

You can deal with roundoff errors by rounding to the nearest integer (see Section 2.2.5) or by displaying a fixed number of digits after the decimal separator (see Section 2.3.2).

Programming Tip 2.2



Do Not Use Magic Numbers

A **magic number** is a numeric constant that appears in your code without explanation. For example,

```
totalVolume = bottles * 2;
```

Why 2? Are bottles twice as voluminous as cans? No, the reason is that every bottle contains 2 liters. Use a named constant to make the code self-documenting:

```
final double BOTTLE_VOLUME = 2;
totalVolume = bottles * BOTTLE_VOLUME;
```

There is another reason for using named constants. Suppose circumstances change, and the bottle volume is now 1.5 liters. If you used a named constant, you make a single change, and you are done. Otherwise, you have to look at every value of 2 in your program and ponder whether it meant a bottle volume, or something else. In a program that is more than a few pages long, that is incredibly tedious and error-prone.

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
final int DAYS_PER_YEAR = 365;
```



© FinBrandy/iStockphoto

We prefer programs that are easy to understand over those that appear to work by magic.

Special Topic 2.1



Numeric Types in Java

In addition to the `int` and `double` types, Java has several other numeric types.

Java has two floating-point types. The `float` type uses half the storage of the `double` type that we use in this book, but it can only store about 7 decimal digits. (In the computer, numbers are represented in the binary number system, using digits 0 and 1.) Many years ago, when computers had far less memory than they have today, `float` was the standard type for floating-point computations, and programmers would indulge in the luxury of “double precision” only when they needed the additional digits. Today, the `float` type is rarely used.

By the way, these numbers are called “floating-point” because of their internal representation in the computer. Consider numbers 29600, 2.96, and 0.0296. They can be represented in a very similar way: namely, as a sequence of the significant digits—296—and an indication of the position of the decimal point. When the values are multiplied or divided by 10, only the

position of the decimal point changes; it “floats”. Computers use base 2, not base 10, but the principle is the same.

In addition to the `int` type, Java has integer types `byte`, `short`, and `long`. Their ranges are shown in Table 4. (Their strange-looking limits are related to powers of 2, another consequence of the fact that computers use binary numbers.)

Table 4 Java Number Types		
Type	Description	Size
<code>int</code>	The integer type, with range –2,147,483,648 (<code>Integer.MIN_VALUE</code>) ... 2,147,483,647 (<code>Integer.MAX_VALUE</code> , about 2.14 billion)	4 bytes
<code>byte</code>	The type describing a single byte consisting of 8 bits, with range –128 ... 127	1 byte
<code>short</code>	The short integer type, with range –32,768 ... 32,767	2 bytes
<code>long</code>	The long integer type, with about 19 decimal digits	8 bytes
<code>double</code>	The double-precision floating-point type, with about 15 decimal digits and a range of about $\pm 10^{308}$	8 bytes
<code>float</code>	The single-precision floating-point type, with about 7 decimal digits and a range of about $\pm 10^{38}$	4 bytes
<code>char</code>	The character type, representing code units in the Unicode encoding scheme (see Computing & Society 2.2)	2 bytes

Special Topic 2.2



Big Numbers

If you want to compute with really large numbers, you can use big number objects. Big number objects are objects of the `BigInteger` and `BigDecimal` classes in the `java.math` package. Unlike the number types such as `int` or `double`, big number objects have essentially no limits on their size and precision. However, computations with big number objects are much slower than those that involve number types. Perhaps more importantly, you can't use the familiar arithmetic operators such as `(+ - *)` with them. Instead, you have to use methods called `add`, `subtract`, and `multiply`. Here is an example of how to create a `BigInteger` object and how to call the `multiply` method:

```
BigInteger oneHundred = new BigInteger("100");
BigInteger fiftyMillion = new BigInteger("50000000");
System.out.println(oneHundred.multiply(fiftyMillion)); // Prints 5000000000
```

The `BigDecimal` type carries out floating-point computations without roundoff errors. For example,

```
BigDecimal price = new BigDecimal("4.35");
BigDecimal quantity = new BigDecimal("100");
BigDecimal total = price.multiply(quantity);
System.out.println(total); // Prints 435.00
```

2.2 Arithmetic

In the following sections, you will learn how to carry out arithmetic calculations in Java.

2.2.1 Arithmetic Operators

© hocus-focus/iStockphoto.



Java supports the same four basic arithmetic operations as a calculator—addition, subtraction, multiplication, and division—but it uses different symbols for multiplication and division.

You must write $a * b$ to denote multiplication. Unlike in mathematics, you cannot write $a \cdot b$, $a \cdot b$, or $a \times b$. Similarly, division is always indicated with $/$, never \div or a fraction bar.

For example, $\frac{a + b}{2}$ becomes $(a + b) / 2$.

The combination of variables, literals, operators, and/or method calls is called an **expression**. For example, $(a + b) / 2$ is an expression.

Parentheses are used just as in algebra: to indicate in which order the parts of the expression should be computed. For example, in the expression $(a + b) / 2$, the sum $a + b$ is computed first, and then the sum is divided by 2. In contrast, in the expression

$a + b / 2$

only b is divided by 2, and then the sum of a and $b / 2$ is formed. As in regular algebraic notation, multiplication and division have a *higher precedence* than addition and subtraction. For example, in the expression $a + b / 2$, the $/$ is carried out first, even though the $+$ operation occurs further to the left.

If you mix integer and floating-point values in an arithmetic expression, the result is a floating-point value. For example, $7 + 4.0$ is the floating-point value 11.0.

Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.

The `++` operator adds 1 to a variable; the `--` operator subtracts 1.

2.2.2 Increment and Decrement

Changing a variable by adding or subtracting 1 is so common that there is a special shorthand for it. The `++` operator increments a variable—see Figure 2:

`counter++; // Adds 1 to the variable counter`

Similarly, the `--` operator decrements a variable:

`counter--; // Subtracts 1 from counter`

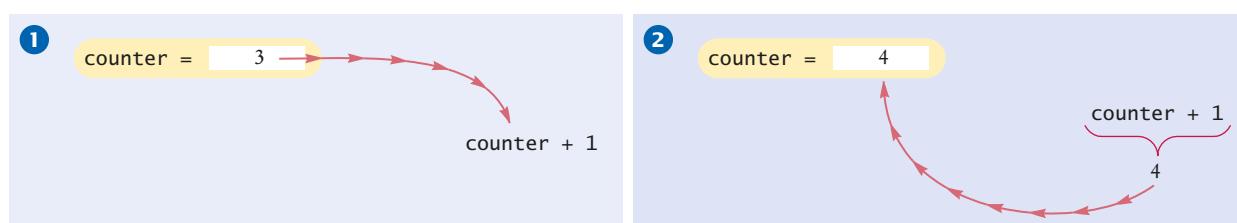


Figure 2 Incrementing a Variable

2.2.3 Integer Division and Remainder

If both arguments of / are integers, the remainder is discarded.

Division works as you would expect, as long as at least one of the numbers involved is a floating-point number. That is,

```
7.0 / 4.0
7 / 4.0
7.0 / 4
```

all yield 1.75. However, if *both* numbers are integers, then the result of the division is always an integer, with the remainder discarded. That is,

```
7 / 4
```

evaluates to 1 because 7 divided by 4 is 1 with a remainder of 3 (which is discarded). This can be a source of subtle programming errors—see Common Error 2.4.

If you are interested in the remainder only, use the % operator:

```
7 % 4
```

is 3, the remainder of the integer division of 7 by 4. The % symbol has no analog in algebra. It was chosen because it looks similar to /, and the remainder operation is related to division. The operator is called **modulus**. (Some people call it *modulo* or *mod*.) It has no relationship with the percent operation that you find on some calculators.

Here is a typical use for the integer / and % operations. Suppose you have an amount of pennies in a piggybank:

```
int pennies = 1729;
```

You want to determine the value in dollars and cents. You obtain the dollars through an integer division by 100:

```
int dollars = pennies / 100; // Sets dollars to 17
```

The integer division discards the remainder. To obtain the remainder, use the % operator:

```
int cents = pennies % 100; // Sets cents to 29
```

See Table 5 for additional examples.



© Michael Flippo/iStockphoto.

Integer division and the % operator yield the dollar and cent values of a piggybank full of pennies.

The % operator computes the remainder of an integer division.

Table 5 Integer Division and Remainder

Expression (where n = 1729)	Value	Comment
n % 10	9	n % 10 is always the last digit of n.
n / 10	172	This is always n without the last digit.
n % 100	29	The last two digits of n.
n / 10.0	172.9	Because 10.0 is a floating-point number, the fractional part is not discarded.
-n % 10	-9	Because the first argument is negative, the remainder is also negative.
n % 2	1	n % 2 is 0 if n is even, 1 or -1 if n is odd.

2.2.4 Powers and Roots

The Java library declares many mathematical functions, such as `Math.sqrt` (square root) and `Math.pow` (raising to a power).

In Java, there are no symbols for powers and roots. To compute them, you must call methods. To take the square root of a number, you use the `Math.sqrt` method. For example, \sqrt{x} is written as `Math.sqrt(x)`. To compute x^n , you write `Math.pow(x, n)`.

In algebra, you use fractions, exponents, and roots to arrange expressions in a compact two-dimensional form. In Java, you have to write all expressions in a linear arrangement. For example, the mathematical expression

$$b \times \left(1 + \frac{r}{100}\right)^n$$

becomes

`b * Math.pow(1 + r / 100, n)`

Figure 3 shows how to analyze such an expression. Table 6 shows additional mathematical methods.

$$\begin{aligned} & b * \text{Math.pow}(1 + r / 100, n) \\ & \quad \underbrace{\qquad\qquad}_{r} \\ & \quad \underbrace{\qquad\qquad}_{100} \\ & \quad \underbrace{\qquad\qquad}_{1 + \frac{r}{100}} \\ & \quad \underbrace{\qquad\qquad}_{\left(1 + \frac{r}{100}\right)^n} \\ & \quad \underbrace{\qquad\qquad}_{b \times \left(1 + \frac{r}{100}\right)^n} \end{aligned}$$

Figure 3
Analyzing an Expression

Table 6 Mathematical Methods

Method	Returns
<code>Math.sqrt(x)</code>	Square root of x (≥ 0)
<code>Math.pow(x, y)</code>	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)
<code>Math.sin(x)</code>	Sine of x (x in radians)
<code>Math.cos(x)</code>	Cosine of x
<code>Math.tan(x)</code>	Tangent of x
<code>Math.toRadians(x)</code>	Convert x degrees to radians (i.e., returns $x \cdot \pi/180$)
<code>Math.toDegrees(x)</code>	Convert x radians to degrees (i.e., returns $x \cdot 180/\pi$)
<code>Math.exp(x)</code>	e^x
<code>Math.log(x)</code>	Natural log ($\ln(x)$, $x > 0$)

Table 6 Mathematical Methods

Method	Returns
<code>Math.log10(x)</code>	Decimal log ($\log_{10}(x)$, $x > 0$)
<code>Math.round(x)</code>	Closest integer to x (as a <code>long</code>)
<code>Math.abs(x)</code>	Absolute value $ x $
<code>Math.max(x, y)</code>	The larger of x and y
<code>Math.min(x, y)</code>	The smaller of x and y

2.2.5 Converting Floating-Point Numbers to Integers

Occasionally, you have a value of type `double` that you need to convert to the type `int`. It is an error to assign a floating-point value to an integer:

```
double balance = total + tax;
int dollars = balance; // Error: Cannot assign double to int
```

You use a cast
(`typeName`) to
convert a value to a
different type.

The compiler disallows this assignment because it is potentially dangerous:

- The fractional part is lost.
- The magnitude may be too large. (The largest integer is about 2 billion, but a floating-point number can be much larger.)

You must use the `cast` operator (`int`) to convert a floating-point value to an integer. Write the cast operator before the expression that you want to convert:

```
double balance = total + tax;
int dollars = (int) balance;
```

The cast (`int`) converts the floating-point value `balance` to an integer by discarding the fractional part. For example, if `balance` is 13.75, then `dollars` is set to 13.

When applying the cast operator to an arithmetic expression, you need to place the expression inside parentheses:

```
int dollars = (int) (total + tax);
```

Syntax 2.4 Cast

Syntax `(typeName) expression`

This is the type of the expression after casting.

These parentheses are a
part of the cast operator.

`(int)` `(balance * 100)`

Use parentheses here if
the cast is applied to an expression
with arithmetic operators.

**FULL CODE EXAMPLE**

Go to wiley.com/go/bj102code to download a program demonstrating casts, rounding, and the % operator.

Discarding the fractional part is not always appropriate. If you want to round a floating-point number to the nearest whole number, use the `Math.round` method. This method returns a `long` integer, because large floating-point numbers cannot be stored in an `int`.

```
long rounded = Math.round(balance);
```

If `balance` is 13.75, then `rounded` is set to 14.

If you know that the result can be stored in an `int` and does not require a `long`, you can use a cast:

```
int rounded = (int) Math.round(balance);
```

Table 7 Arithmetic Expressions

Mathematical Expression	Java Expression	Comments
$\frac{x + y}{2}$	<code>(x + y) / 2</code>	The parentheses are required; <code>x + y / 2</code> computes $x + \frac{y}{2}$.
$\frac{xy}{2}$	<code>x * y / 2</code>	Parentheses are not required; operators with the same precedence are evaluated left to right.
$\left(1 + \frac{r}{100}\right)^n$	<code>Math.pow(1 + r / 100, n)</code>	Use <code>Math.pow(x, n)</code> to compute x^n .
$\sqrt{a^2 + b^2}$	<code>Math.sqrt(a * a + b * b)</code>	<code>a * a</code> is simpler than <code>Math.pow(a, 2)</code> .
$\frac{i + j + k}{3}$	<code>(i + j + k) / 3.0</code>	If i, j , and k are integers, using a denominator of 3.0 forces floating-point division.
π	<code>Math.PI</code>	<code>Math.PI</code> is a constant declared in the <code>Math</code> class.

SELF CHECK

10. A bank account earns interest once per year. In Java, how do you compute the interest earned in the first year? Assume variables `percent` and `balance` of type `double` have already been declared.
11. In Java, how do you compute the side length of a square whose area is stored in the variable `area`?
12. The volume of a sphere is given by

$$V = \frac{4}{3}\pi r^3$$

If the radius is given by a variable `radius` of type `double`, write a Java expression for the volume.

13. What is the value of `1729 / 10` and `1729 % 10`?
14. If `n` is a positive integer, what is `(n / 10) % 10`?

Practice It

Now you can try these exercises at the end of the chapter: R2.4, R2.6, E2.4, P2.6.

Common Error 2.4



Unintended Integer Division

It is unfortunate that Java uses the same symbol, namely `/`, for both integer and floating-point division. These are really quite different operations. It is a common error to use integer division by accident. Consider this segment that computes the average of three integers.

```
int score1 = 10;
int score2 = 4;
int score3 = 9;
double average = (score1 + score2 + score3) / 3; // Error
System.out.println("Average score: " + average); // Prints 7.0, not 7.666666666666667
```

What could be wrong with that? Of course, the average of `score1`, `score2`, and `score3` is

$$\frac{\text{score1} + \text{score2} + \text{score3}}{3}$$

Here, however, the `/` does not mean division in the mathematical sense. It denotes integer division because both `score1 + score2 + score3` and `3` are integers. Because the scores add up to 23, the average is computed to be 7, the result of the integer division of 23 by 3. That integer 7 is then moved into the floating-point variable `average`. The remedy is to make the numerator or denominator into a floating-point number:

```
double total = score1 + score2 + score3;
double average = total / 3;
```

or

```
double average = (score1 + score2 + score3) / 3.0;
```

Common Error 2.5



Unbalanced Parentheses

Consider the expression

$$((a + b) * t) / 2 * (1 - t)$$

What is wrong with it? Count the parentheses. There are three `(` and two `)`. The parentheses are *unbalanced*. This kind of typing error is very common with complicated expressions. Now consider this expression.

$$(a + b) * t) / (2 * (1 - t)$$

This expression has three `(` and three `)`, but it still is not correct. In the middle of it,

$$(a + b) * t) / (2 * (1 - t)$$

↑



© Crokof/iStockphoto.

there is only one `(` but two `)`, which is an error. In the middle of an expression, the count of `(` must be greater than or equal to the count of `)`, and at the end of the expression the two counts must be the same.

Here is a simple trick to make the counting easier without using pencil and paper. It is difficult for the brain to keep two counts simultaneously. Keep only one count when scanning the expression. Start with 1 at the first opening parenthesis, add 1 whenever you see an opening parenthesis, and subtract one whenever you see a closing parenthesis. Say the numbers aloud as you scan the expression. If the count ever drops below zero, or is not zero at the end, the parentheses are unbalanced. For example, when scanning the previous expression, you would mutter

$$(a + b) * t) / (2 * (1 - t)$$

1 0 -1

and you would find the error.

Programming Tip 2.3**Spaces in Expressions**

It is easier to read

```
x1 = (-b + Math.sqrt(b * b - 4 * a * c)) / (2 * a);
```

than

```
x1=(-b+Math.sqrt(b*b-4*a*c))/(2*a);
```

Simply put spaces around all operators + - * / %. However, don't put a space after a *unary* minus: a – used to negate a single quantity, such as -b. That way, it can be easily distinguished from a *binary* minus, as in a - b.

It is customary not to put a space after a method name. That is, write `Math.sqrt(x)` and not `Math.sqrt (x)`.

Java 8 Note 2.1**Avoiding Negative Remainders**

The % operator yields negative values when the first operand is negative. This can be an annoyance. For example, suppose a robot keeps track of directions in degrees between 0 and 359. Now the robot turns by some number of degrees. You can't simply compute the new direction as `(direction + turn) % 360` because you might get a negative result (see Exercise R2.9). In Java 8, you can instead call

```
Math.floorMod(direction + turn, 360)
```

to compute the correct remainder. The result of `Math.floorMod(m, n)` is always positive when n is positive.

Special Topic 2.3**Combining Assignment and Arithmetic**

In Java, you can combine arithmetic and assignment. For example, the instruction

```
total += cans;
```

is a shortcut for

```
total = total + cans;
```

Similarly,

```
total *= 2;
```

is another way of writing

```
total = total * 2;
```

Many programmers find this a convenient shortcut. If you like it, go ahead and use it in your own code. For simplicity, we won't use it in this book, though.

**VIDEO EXAMPLE 2.1****Using Integer Division**

A punch recipe calls for a given amount of orange soda. See how to compute the required number of 12-ounce cans, using integer division. Go to wiley.com/go/bjlo2videos to view Video Example 2.1.



© Maxfocus/Stockphoto.



Computing & Society 2.1 The Pentium Floating-Point Bug

In 1994, Intel Corporation released what was then its most powerful processor, the Pentium. Unlike previous generations of its processors, it had a very fast floating-point unit. Intel's goal was to compete aggressively with the makers of higher-end processors for engineering workstations. The Pentium was a huge success immediately.

In the summer of 1994, Dr. Thomas Nicely of Lynchburg College in Virginia ran an extensive set of computations to analyze the sums of reciprocals of certain sequences of prime numbers. The results were not always what his theory predicted, even after he took into account the inevitable roundoff errors. Then Dr. Nicely noted that the same program did produce the correct results when running on the slower 486 processor that preceded the Pentium in Intel's lineup. This should not have happened. The optimal round-off behavior of floating-point calculations has been standardized by the Institute for Electrical and Electronic Engineers (IEEE) and Intel claimed to adhere to the IEEE standard in both the 486 and the Pentium processors. Upon further checking, Dr. Nicely discovered that indeed there was a very small set of numbers for which the product of two numbers was computed differently on the two processors. For example,

$$4,195,835 - ((4,195,835/3,145,727) \times 3,145,727)$$

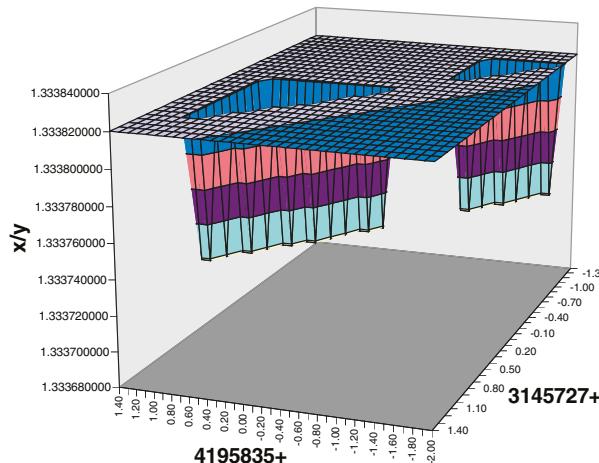
is mathematically equal to 0, and it did compute as 0 on a 486 processor. On his Pentium processor the result was 256.

As it turned out, Intel had independently discovered the bug in its testing and had started to produce chips that fixed it. The bug was caused by an error in a table that was used to speed up the floating-point multiplication algorithm of the processor. Intel determined that the problem was exceedingly rare. They claimed that under normal use, a typical consumer would only notice the problem once every 27,000 years. Unfortunately for Intel, Dr. Nicely had not been a normal user.

Now Intel had a real problem on its hands. It figured that the cost of replacing all Pentium processors that it had sold so far would cost a great deal of money. Intel already had more orders for the chip than it could produce, and it would be particularly galling to have to give out the scarce chips as free replacements instead of selling them. Intel's management decided to punt on the issue and initially offered to replace the processors only for those customers who could prove that their work required absolute precision in mathematical calculations. Naturally, that did not go over well with the hundreds of thousands of customers who had paid retail prices of \$700 and more for a Pentium chip and did not want to live with the nagging feeling that perhaps, one day, their income tax program would produce a faulty return.

Ultimately, Intel caved in to public demand and replaced all defective chips, at a cost of about 475 million dollars.

Pentium FDIV error



This graph shows a set of numbers for which the original Pentium processor obtained the wrong quotient.

Courtesy of Larry Hoyle, Institute for Policy & Social Research,
University of Kansas.

2.3 Input and Output

In the following sections, you will see how to read user input and how to control the appearance of the output that your programs produce.

2.3.1 Reading Input

You can make your programs more flexible if you ask the program user for inputs rather than using fixed values. Consider, for example, a program that processes prices



© Media Bakery.

A supermarket scanner reads bar codes. The Java Scanner reads numbers and text.

Java classes are grouped into packages. Use the `import` statement to use classes from packages.

and quantities of soda containers. Prices and quantities are likely to fluctuate. The program user should provide them as inputs.

When a program asks for user input, it should first print a message that tells the user which input is expected. Such a message is called a **prompt**.

```
System.out.print("Please enter the number of bottles: "); // Display prompt
```

Use the `print` method, not `println`, to display the prompt. You want the input to appear after the colon, not on the following line. Also remember to leave a space after the colon.

Because output is sent to `System.out`, you might think that you use `System.in` for input. Unfortunately, it isn't quite that simple. When Java was first designed, not much attention was given to reading keyboard input. It was assumed that all programmers would produce graphical user interfaces with text fields and menus. `System.in` was given a minimal set of features and must be combined with other classes to be useful.

To read keyboard input, you use a class called `Scanner`. You obtain a `Scanner object` by using the following statement:

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

You will learn more about objects and classes in Chapter 8. For now, simply include this statement whenever you want to read keyboard input.

When using the `Scanner` class, you need to carry out another step: import the class from its **package**. A package is a collection of classes with a related purpose. All classes in the Java library are contained in packages. The `System` class belongs to the package `java.lang`. The `Scanner` class belongs to the package `java.util`.

Only the classes in the `java.lang` package are automatically available in your programs. To use the `Scanner` class from the `java.util` package, place the following declaration at the top of your program file:

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

Once you have a scanner, you use its `nextInt` method to read an integer value:

```
System.out.print("Please enter the number of bottles: ");
int bottles = in.nextInt();
```

Syntax 2.5 Input Statement

Include this line so you can use the `Scanner` class.

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

Create a `Scanner` object to read keyboard input.

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

Don't use `println` here.

Display a prompt in the console window.

```
System.out.print("Please enter the number of bottles: ");
```

Define a variable to hold the input value.

```
int bottles = in.nextInt();
```

The program waits for user input, then places the input into the variable.

Use the Scanner class to read keyboard input in a console window.

When the `nextInt` method is called, the program waits until the user types a number and presses the Enter key. After the user supplies the input, the number is placed into the `bottles` variable, and the program continues.

To read a floating-point number, use the `nextDouble` method instead:

```
System.out.print("Enter price: ");
double price = in.nextDouble();
```

2.3.2 Formatted Output

When you print the result of a computation, you often want to control its appearance. For example, when you print an amount in dollars and cents, you usually want it to be rounded to two significant digits. That is, you want the output to look like

Price per liter: 1.22

instead of

Price per liter: 1.215962441314554

The following command displays the price with two digits after the decimal point:

```
System.out.printf("%.2f", price);
```

You can also specify a *field width*:

```
System.out.printf("%10.2f", price);
```

The price is printed using ten characters: six spaces followed by the four characters 1.22.

The construct `%10.2f` is called a *format specifier*: it describes how a value should be formatted. The letter `f` at the end of the format specifier indicates that we are displaying a floating-point number. Use `d` for an integer and `s` for a string; see Table 8 for examples.

Table 8 Format Specifier Examples

Format String	Sample Output	Comments
"%d"	24	Use <code>d</code> with an integer.
"%5d"	24	Spaces are added so that the field width is 5.
"Quantity:%5d"	Quantity: 24	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1.21997	Use <code>f</code> with a floating-point number.
"%.2f"	1.22	Prints two digits after the decimal point.
"%7.2f"	1.22	Spaces are added so that the field width is 7.
"%s"	Hello	Use <code>s</code> with a string.
"%d %.2f"	24 1.22	You can format multiple values at once.
"Hello%nWorld%n"	Hello World	Each <code>\n</code> causes subsequent output to continue on a new line.

You use the `printf` method to line up your output in neat columns.

NAME	COMMENCEMENT			TERM	EXPIRATION	
	MONTH	DAY	YEAR		MONTH	DAY
Orneward	June	4	1920	6 yrs	June	14
Orneward	April	24	1920	5 yrs	April	24
Slager	Mar	14	1943	5 yrs	Mar	14
Slager	Feb	9	1920	old	Mar	14
Slager	Oct	20	1920	5 yrs	Oct	20
C. W. Hamill Co. S. Boer	Mar	15	1924	5 yrs	Mar	15
Quin	Nov	14	1924	5 yrs	Nov	14
Sr.	Sept	5	1920	5 yrs	Sept	5
Dodge	May	22	1920	5 yrs	May	22
T	March	1925			Oct	25

© Koelle/Stockphoto

A format string contains format specifiers and literal characters. Any characters that are not format specifiers are printed verbatim. For example, the command

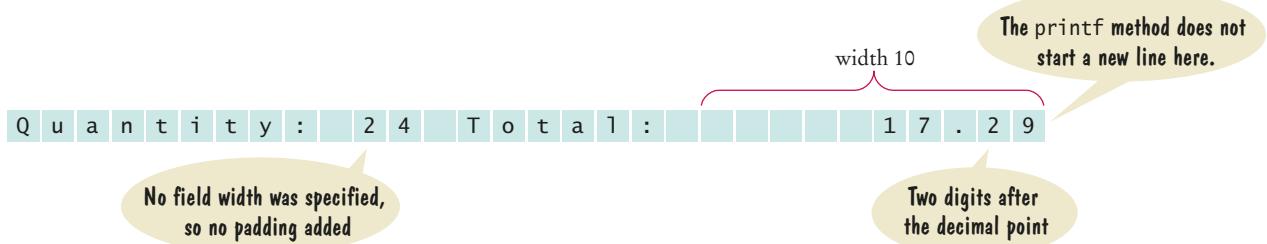
```
System.out.printf("Price per liter:%10.2f", price);
```

prints

```
Price per liter: 1.22
```

You can print multiple values with a single call to the `printf` method. Here is a typical example:

```
System.out.printf("Quantity: %d Total: %10.2f", quantity, total);
```



The `printf` method, like the `print` method, does not start a new line after the output. If you want output to continue on a separate line, add the `\n` format specifier. For example, the statement

```
System.out.printf("Quantity: %7d\nTotal: %10.2f\n", quantity, total);
```

yields a printout

```
Quantity: 24
Total: 17.29
```

Our next example program will prompt for the price of a six-pack and the volume of each can, then print out the price per ounce. The program puts to work what you just learned about reading input and formatting output.

sec03/Volume2.java

```
1 import java.util.Scanner;
2
3 /**
4  * This program prints the price per ounce for a six-pack of cans.
5 */
6 public class Volume2
7 {
```

```

8  public static void main(String[] args)
9  {
10 // Read price per pack
11 Scanner in = new Scanner(System.in);
12
13 System.out.print("Please enter the price for a six-pack: ");
14 double packPrice = in.nextDouble();
15
16 // Read can volume
17
18 System.out.print("Please enter the volume for each can (in ounces): ");
19 double canVolume = in.nextDouble();
20
21 // Compute pack volume
22
23 final double CANS_PER_PACK = 6;
24 double packVolume = canVolume * CANS_PER_PACK;
25
26 // Compute and print price per ounce
27
28 double pricePerOunce = packPrice / packVolume;
29
30 System.out.printf("Price per ounce: %.2f", pricePerOunce);
31 System.out.println();
32 }
33 }
34 }
```

Program Run

```

Please enter the price for a six-pack: 2.95
Please enter the volume for each can (in ounces): 12
Price per ounce: 0.04
```

SELF CHECK

15. Write statements to prompt for and read the user's age using a Scanner variable named `in`.
16. What is wrong with the following statement sequence?

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextDouble();
int quantity = in.nextInt();
```
17. What is problematic about the following statement sequence?

```
System.out.print("Please enter the unit price: ");
double unitPrice = in.nextInt();
```
18. What is problematic about the following statement sequence?

```
System.out.print("Please enter the number of cans");
int cans = in.nextInt();
```
19. What is the output of the following statement sequence?

```
int volume = 10;
System.out.printf("The volume is %5d", volume);
```
20. Using the `printf` method, print the values of the integer variables `bottles` and `cans` so that the output looks like this:
Bottles: 8
Cans: 24

The numbers to the right should line up. (You may assume that the numbers have at most 8 digits.)

Practice It Now you can try these exercises at the end of the chapter: R2.13, E2.6, E2.7.

Programming Tip 2.4



Use the API Documentation

The classes and methods of the Java library are listed in the **API documentation**. The API is the “**application programming interface**”. A programmer who uses the Java classes to put together a computer program (or *application*) is an *application programmer*. That’s you. In contrast, the programmers who designed and implemented the library classes (such as `Scanner`) are *system programmers*.

The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

You can find the API documentation at <http://docs.oracle.com/javase/8/docs/api/index.html>. The API documentation describes all classes in the Java library—there are thousands of them. Fortunately, only a few are of interest to the beginning programmer. To learn more about a class, click on its name in the left hand column. You can then find out the package to which the class belongs, and which methods it supports (see Figure 4). Click on the link of a method to get a detailed description.

Appendix D contains an abbreviated version of the API documentation.

The screenshot shows a Mozilla Firefox browser window displaying the Java API documentation for the `Scanner` class. The URL in the address bar is <https://docs.oracle.com/javase/8/docs/api/>. The page title is "Scanner (Java Platform SE 8) - Mozilla Firefox". The navigation menu at the top includes OVERVIEW, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, DEPRECATED, INDEX, and HELP. A sub-menu for the CLASS tab shows PREV CLASS, NEXT CLASS, FRAMES, and NO FRAMES. Below the menu, there are links for SUMMARY, NESTED, FIELD, CONSTR, and METHOD, and DETAIL, FIELD, CONSTR, and METHOD. The main content area shows the `Scanner` class definition under the heading "Class Scanner". It extends `Object` and implements `Iterator<String>` and `Closeable`. The class is described as a simple text scanner. A code example shows how to use the `Scanner` class to read from `System.in`:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Figure 4 The API Documentation of the Standard Java Library

HOW TO 2.1**Carrying out Computations**

Many programming problems require arithmetic computations. This How To shows you how to turn a problem statement into pseudocode and, ultimately, a Java program.

Problem Statement Suppose you are asked to write a program that simulates a vending machine. A customer selects an item for purchase and inserts a bill into the vending machine. The vending machine dispenses the purchased item and gives change. We will assume that all item prices are multiples of 25 cents, and the machine gives all change in dollar coins and quarters. Your task is to compute how many coins of each type to return.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there are two inputs:

- The denomination of the bill that the customer inserts
- The price of the purchased item

There are two desired outputs:

- The number of dollar coins that the machine returns
- The number of quarters that the machine returns

Step 2 Work out examples by hand.

This is a very important step. If you can't compute a couple of solutions by hand, it's unlikely that you'll be able to write a program that automates the computation.

Let's assume that a customer purchased an item that cost \$2.25 and inserted a \$5 bill. The customer is due \$2.75, or two dollar coins and three quarters, in change.

That is easy for you to see, but how can a Java program come to the same conclusion? The key is to work in pennies, not dollars. The change due the customer is 275 pennies. Dividing by 100 yields 2, the number of dollars. Dividing the remainder (75) by 25 yields 3, the number of quarters.

Step 3 Write pseudocode for computing the answers.

In the previous step, you worked out a specific instance of the problem. You now need to come up with a method that works in general.

Given an arbitrary item price and payment, how can you compute the coins due? First, compute the change due in pennies:

change due = 100 x bill value - item price in pennies

To get the dollars, divide by 100 and discard the remainder:

dollar coins = change due / 100 (without remainder)

The remaining change due can be computed in two ways. If you are familiar with the modulus operator, you can simply compute

change due = change due % 100

Alternatively, subtract the penny value of the dollar coins from the change due:

change due = change due - 100 x dollar coins

To get the quarters due, divide by 25:

quarters = change due / 25

Step 4 Declare the variables and constants that you need, and specify their types.

Here, we have five variables:

- billValue
- itemPrice
- changeDue
- dollarCoins
- quarters

Should we introduce constants to explain 100 and 25 as PENNIES_PER_DOLLAR and PENNIES_PER_QUARTER? Doing so will make it easier to convert the program to international markets, so we will take this step.

It is very important that changeDue and PENNIES_PER_DOLLAR are of type `int` because the computation of dollarCoins uses integer division. Similarly, the other variables are integers.

Step 5 Turn the pseudocode into Java statements.

If you did a thorough job with the pseudocode, this step should be easy. Of course, you have to know how to express mathematical operations (such as powers or integer division) in Java.

```
changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
dollarCoins = changeDue / PENNIES_PER_DOLLAR;
changeDue = changeDue % PENNIES_PER_DOLLAR;
quarters = changeDue / PENNIES_PER_QUARTER;
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the bill value and item price:

```
System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
billValue = in.nextInt();
System.out.print("Enter item price in pennies: ");
itemPrice = in.nextInt();
```

When the computation is finished, we display the result. For extra credit, we use the `printf` method to make sure that the output lines up neatly.

```
System.out.printf("Dollar coins: %6d", dollarCoins);
System.out.printf("Quarters:      %6d", quarters);
```

Step 7 Provide a class with a `main` method.

Your computation needs to be placed into a class. Find an appropriate name for the class that describes the purpose of the computation. In our example, we will choose the name `VendingMachine`.

Inside the class, supply a `main` method.



A vending machine takes bills and gives change in coins.

In the `main` method, you need to declare constants and variables (Step 4), carry out computations (Step 5), and provide input and output (Step 6). Clearly, you will want to first get the input, then do the computations, and finally show the output. Declare the constants at the beginning of the method, and declare each variable just before it is needed.

Here is the complete program, `how_to_1/VendingMachine.java`:

```
import java.util.Scanner;

/**
 * This program simulates a vending machine that gives change.
 */
public class VendingMachine
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);

        final int PENNIES_PER_DOLLAR = 100;
        final int PENNIES_PER_QUARTER = 25;

        System.out.print("Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): ");
        int billValue = in.nextInt();
        System.out.print("Enter item price in pennies: ");
        int itemPrice = in.nextInt();

        // Compute change due

        int changeDue = PENNIES_PER_DOLLAR * billValue - itemPrice;
        int dollarCoins = changeDue / PENNIES_PER_DOLLAR;
        changeDue = changeDue % PENNIES_PER_DOLLAR;
        int quarters = changeDue / PENNIES_PER_QUARTER;

        // Print change due

        System.out.printf("Dollar coins: %6d", dollarCoins);
        System.out.println();
        System.out.printf("Quarters:      %6d", quarters);
        System.out.println();
    }
}
```

Program Run

```
Enter bill value (1 = $1 bill, 5 = $5 bill, etc.): 5
Enter item price in pennies: 225
Dollar coins:      2
Quarters:         3
```



WORKED EXAMPLE 2.1

Computing the Cost of Stamps



Learn how to use arithmetic functions to simulate a stamp vending machine. Go to wiley.com/go/bjlo2examples and download Worked Example 2.1.

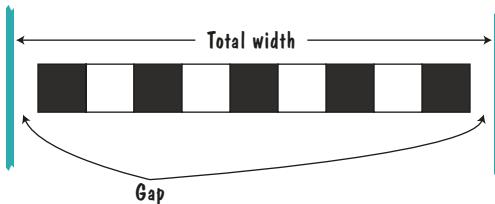
2.4 Problem Solving: First Do It By Hand

A very important step for developing an algorithm is to first carry out the computations *by hand*. If you can't compute a solution yourself, it's unlikely that you'll be able to write a program that automates the computation.

To illustrate the use of hand calculations, consider the following problem.

A row of black and white tiles needs to be placed along a wall. For aesthetic reasons, the architect has specified that the first and last tile shall be black.

Your task is to compute the number of tiles needed and the gap at each end, given the space available and the width of each tile.



Pick concrete values
for a typical situation
to use in a hand
calculation.

To make the problem more concrete, let's assume the following dimensions:

- Total width: 100 inches
- Tile width: 5 inches

The obvious solution would be to fill the space with 20 tiles, but that would not work—the last tile would be white.

Instead, look at the problem this way: The first tile must always be black, and then we add some number of white/black pairs:



The first tile takes up 5 inches, leaving 95 inches to be covered by pairs. Each pair is 10 inches wide. Therefore the number of pairs is $95 / 10 = 9.5$. However, we need to discard the fractional part since we can't have fractions of tile pairs.

Therefore, we will use 9 tile pairs or 18 tiles, plus the initial black tile. Altogether, we require 19 tiles.

The tiles span $19 \times 5 = 95$ inches, leaving a total gap of $100 - 95 = 5$ inches.

The gap should be evenly distributed at both ends. At each end, the gap is $(100 - 95) / 2 = 2.5$ inches.

This computation gives us enough information to devise an algorithm with arbitrary values for the total width and tile width.

$$\text{number of pairs} = \text{integer part of } (\text{total width} - \text{tile width}) / (2 \times \text{tile width})$$

$$\text{number of tiles} = 1 + 2 \times \text{number of pairs}$$

$$\text{gap at each end} = (\text{total width} - \text{number of tiles} \times \text{tile width}) / 2$$

FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a program that implements this algorithm.

As you can see, doing a hand calculation gives enough insight into the problem that it becomes easy to develop an algorithm.

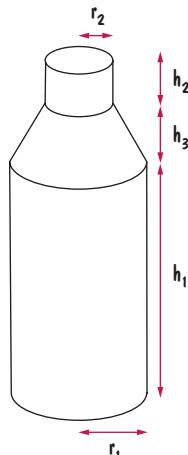
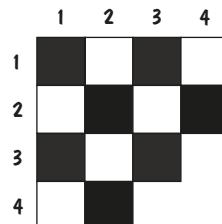


- 21.** Translate the pseudocode for computing the number of tiles and the gap width into Java.
- 22.** Suppose the architect specifies a pattern with black, gray, and white tiles, like this:



Again, the first and last tile should be black. How do you need to modify the algorithm?

- 23.** A robot needs to tile a floor with alternating black and white tiles. Develop an algorithm that yields the color (0 for black, 1 for white), given the row and column number. Start with specific values for the row and column, and then generalize.



- 24.** For a particular car, repair and maintenance costs in year 1 are estimated at \$100; in year 10, at \$1,500. Assuming that the repair cost increases by the same amount every year, develop pseudocode to compute the repair cost in year 3 and then generalize to year n .
- 25.** The shape of a bottle is approximated by two cylinders of radius r_1 and r_2 and heights h_1 and h_2 , joined by a cone section of height h_3 .
Using the formulas for the volume of a cylinder, $V = \pi r^2 h$, and a cone section,

$$V = \pi \frac{(r_1^2 + r_1 r_2 + r_2^2)h}{3},$$

develop pseudocode to compute the volume of the bottle. Using an actual bottle with known volume as a sample, make a hand calculation of your pseudocode.

Practice It Now you can try these exercises at the end of the chapter: R2.18, R2.20, R2.21.



WORKED EXAMPLE 2.2

Computing Travel Time



Learn how to develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain. Go to wiley.com/go/bjlo2examples and download Worked Example 2.2.



Courtesy of NASA.

2.5 Strings

Strings are sequences of characters.

Many programs process text, not numbers. Text consists of **characters**: letters, numbers, punctuation, spaces, and so on. A **string** is a sequence of characters. For example, the string "Harry" is a sequence of five characters.



© essxboy/istockphoto.com

2.5.1 The String Type

You can declare variables that hold strings.

```
String name = "Harry";
```

We distinguish between string variables (such as the variable `name` declared above) and string **literals** (character sequences enclosed in quotes, such as "Harry"). A string variable is simply a variable that can hold a string, just as an integer variable can hold an integer. A string literal denotes a particular string, just as a number literal (such as 2) denotes a particular number.

The number of characters in a string is called the *length* of the string. For example, the length of "Harry" is 5. You can compute the length of a string with the `length` method.

```
int n = name.length();
```

A string of length 0 is called the *empty string*. It contains no characters and is written as "".

The `length` method yields the number of characters in a string.

Use the + operator to *concatenate* strings; that is, to put them together to yield a longer string.

2.5.2 Concatenation

Given two strings, such as "Harry" and "Morgan", you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Java, you use the + operator to concatenate two strings.

For example,

```
String fName = "Harry";
String lName = "Morgan";
String name = fName + lName;
```

results in the string

"HarryMorgan"

What if you'd like the first and last name separated by a space? No problem:

```
String name = fName + " " + lName;
```

This statement concatenates three strings: `fName`, the string literal " ", and `lName`. The result is

"Harry Morgan"

When the expression to the left or the right of a + operator is a string, the other one is automatically forced to become a string as well, and both strings are concatenated.

Whenever one of the arguments of the + operator is a string, the other argument is converted to a string.

For example, consider this code:

```
String jobTitle = "Agent";
int employeeId = 7;
String bond = jobTitle + employeeId;
```

Because jobTitle is a string, employeeId is converted from the integer 7 to the string "7". Then the two strings "Agent" and "7" are concatenated to form the string "Agent7".

This concatenation is very useful for reducing the number of System.out.print instructions. For example, you can combine

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

to the single call

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

The concatenation "The total is " + total computes a single string that consists of the string "The total is ", followed by the string equivalent of the number total.

2.5.3 String Input

Use the next method of the Scanner class to read a string containing a single word.

You can read a string from the console:

```
System.out.print("Please enter your name: ");
String name = in.next();
```

When a string is read with the next method, only one word is read. For example, suppose the user types

Harry Morgan

as the response to the prompt. This input consists of two words. The call in.next() yields the string "Harry". You can use another call to in.next() to read the second word.

2.5.4 Escape Sequences

To include a quotation mark in a literal string, precede it with a backslash (\), like this:

```
"He said \"Hello\""
```

The backslash is not included in the string. It indicates that the quotation mark that follows should be a part of the string and not mark the end of the string. The sequence \" is called an **escape sequence**.

To include a backslash in a string, use the escape sequence \\, like this:

```
"C:\\Temp\\Secret.txt"
```

Another common escape sequence is \n, which denotes a **newline** character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
System.out.print("*\\n**\\n***\\n");
```

prints the characters

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

on three separate lines.

However, in Windows, you need to add a “return” character \r before each \n. If you use the %n format specifier in a call to `System.out.printf`, it is automatically translated into \n or \r\n:

```
System.out.printf("Price: %10.2f%n", price);
```

2.5.5 Strings and Characters

Strings are sequences of Unicode characters (see Computing & Society 2.2). In Java, a **character** is a value of the type `char`. Characters have numeric values. You can find the values of the characters that are used in Western European languages in Appendix A. For example, if you look up the value for the character 'H', you can see that is actually encoded as the number 72.

Character literals are delimited by single quotes, and you should not confuse them with strings.

- 'H' is a character, a value of type `char`.
- "H" is a string containing a single character, a value of type `String`.

String positions are counted starting with 0.

The `charAt` method returns a `char` value from a string. The first string position is labeled 0, the second one 1, and so on.

H	a	r	r	y
0	1	2	3	4

The position number of the last character (4 for the string "Harry") is always one less than the length of the string.

For example, the statement

```
String name = "Harry";
char start = name.charAt(0);
char last = name.charAt(4);
```

sets `start` to the value 'H' and `last` to the value 'y'.

2.5.6 Substrings

Use the `substring` method to extract a part of a string.

Once you have a string, you can extract substrings by using the `substring` method. The method call

```
str.substring(start, pastEnd)
```

returns a string that is made up of the characters in the string `str`, starting at position `start`, and containing all characters up to, but not including, the position `pastEnd`. Here is an example:

```
String greeting = "Hello, World!";
String sub = greeting.substring(0, 5); // sub is "Hello"
```

The `substring` operation makes a string that consists of the first five characters taken from the string `greeting`.

H	e	l	l	o	,	W	o	r	l	d	!	
0	1	2	3	4	5	6	7	8	9	10	11	12

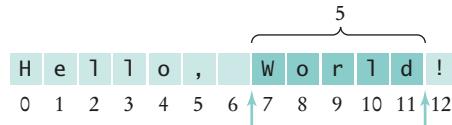


© slpixx/
istockphoto.

A string is a sequence of characters.

Let's figure out how to extract the substring "World". Count characters starting at 0, not 1. You find that `w` has position number 7. The first character that you don't want, `!`, is the character at position 12. Therefore, the appropriate substring command is

```
String sub2 = greeting.substring(7, 12);
```



It is curious that you must specify the position of the first character that you do want and then the first character that you don't want. There is one advantage to this setup. You can easily compute the length of the substring: It is `pastEnd - start`. For example, the string "World" has length $12 - 7 = 5$.

If you omit the end position when calling the `substring` method, then all characters from the starting position to the end of the string are copied. For example,

```
String tail = greeting.substring(7); // Copies all characters from position 7 on
sets tail to the string "World!".
```

Following is a simple program that puts these concepts to work. The program asks for your name and that of your significant other. It then prints out your initials.

The operation `first.substring(0, 1)` makes a string consisting of one character, taken from the start of `first`. The program does the same for the second. Then it concatenates the resulting one-character strings with the string literal "&" to get a string of length 3, the initials string. (See Figure 5.)

```
first = R o d o l f o
        0 1 2 3 4 5 6
second = S a l l y
         0 1 2 3 4

initials = R & S
          0 1 2
```

Figure 5 Building the initials String



© Rich Legg/iStockphoto.

Initials are formed from the first letter of each name.

sec05/Initials.java

```
1 import java.util.Scanner;
2
3 /**
4  * This program prints a pair of initials.
5 */
6 public class Initials
7 {
8     public static void main(String[] args)
9     {
10        Scanner in = new Scanner(System.in);
11    }
}
```

```

12 // Get the names of the couple
13
14 System.out.print("Enter your first name: ");
15 String first = in.next();
16 System.out.print("Enter your significant other's first name: ");
17 String second = in.next();
18
19 // Compute and display the inscription
20
21 String initials = first.substring(0, 1)
22     + "&" + second.substring(0, 1);
23 System.out.println(initials);
24 }
25 }
```

Program Run

```

Enter your first name: Rodolfo
Enter your significant other's first name: Sally
R&S
```

Table 9 String Operations

Statement	Result	Comment
string str = "Ja"; str = str + "va";	str is set to "Java"	When applied to strings, + denotes concatenation.
System.out.println("Please" + " enter your name: ");	Prints Please enter your name:	Use concatenation to break up strings that don't fit into one line.
team = 49 + "ers";	team is set to "49ers"	Because "ers" is a string, 49 is converted to a string.
String first = in.next(); String last = in.next(); (User input: Harry Morgan)	first contains "Harry" last contains "Morgan"	The next method places the next word into the string variable.
String greeting = "H & S"; int n = greeting.length();	n is set to 5	Each space counts as one character.
String str = "Sally"; char ch = str.charAt(1);	ch is set to 'a'	This is a char value, not a String. Note that the initial position is 0.
String str = "Sally"; String str2 = str.substring(1, 4);	str2 is set to "all"	Extracts the substring starting at position 1 and ending before position 4.
String str = "Sally"; String str2 = str.substring(1);	str2 is set to "ally"	If you omit the end position, all characters from the position until the end of the string are included.
String str = "Sally"; String str2 = str.substring(1, 2);	str2 is set to "a"	Extracts a String of length 1; contrast with str.charAt(1).
String last = str.substring(str.length() - 1);	last is set to the string containing the last character in str	The last character has position str.length() - 1.


SELF CHECK


26. What is the length of the string "Java Program"?
27. Consider this string variable.

```
String str = "Java Program";
```

 Give a call to the `substring` method that returns the substring "gram".
28. Use string concatenation to turn the string variable `str` from Self Check 27 into "Java Programming".
29. What does the following statement sequence print?

```
String str = "Harry";
int n = str.length();
String mystery = str.substring(0, 1) + str.substring(n - 1, n);
System.out.println(mystery);
```
30. Give an input statement sequence to read a name of the form "John Q. Public".

Practice It Now you can try these exercises at the end of the chapter: R2.8, R2.14, E2.15, P2.4.

Special Topic 2.4



Instance Methods and Static Methods

In this chapter, you have learned how to read, process, and print numbers and strings. Many of these tasks involve various method calls. You may have noticed syntactical differences in these method calls. For example, to compute the square root of a number `num`, you call `Math.sqrt(num)`, but to compute the length of a string `str`, you call `str.length()`. This section explains the reasons behind these differences.

The Java language distinguishes between values of **primitive types** and **objects**. Numbers and characters, as well as the values `false` and `true` that you will see in Chapter 3, are primitive. All other values are objects. Examples of objects are

- a string such as "Hello".
- a `Scanner` object obtained by calling `in = new Scanner(System.in)`.
- `System.in` and `System.out`.

In Java, each object belongs to a **class**. For example,

- All strings are objects of the `String` class.
- A `scanner` object belongs to the `Scanner` class.
- `System.out` is an object of the `PrintStream` class. (It is useful to know this so that you can look up the valid methods in the API documentation; see Programming Tip 2.4 on page 55.)

A class declares the methods that you can use with its objects. Here are examples of methods that are invoked on objects:

```
"Hello".substring(0, 1)
in.nextDouble()
System.out.println("Hello")
```

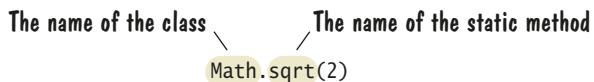
A method is invoked with the **dot notation**: the object is followed by the name of the method, and the method is followed by parameters enclosed in parentheses.

The method is invoked on this object. This is the name of the method. These parameters are inputs to the method.

System.out.println("Hello")

You cannot invoke methods on numbers. For example, the call `2.sqrt()` would be an error.

In Java, classes can declare methods that are *not* invoked on objects. Such methods are called **static methods**. (The term “static” is a historical holdover from the C and C++ programming languages. It has nothing to do with the usual meaning of the word.) For example, the `Math` class declares a static method `sqrt`. You call it by giving the name of the class and method, then the name of the numeric input: `Math.sqrt(2)`.



In contrast, a method that is invoked on an object is called an **instance method**. As a rule of thumb, you use static methods when you manipulate numbers. You use instance methods when you process strings or perform input/output. You will learn more about the distinction between static and instance methods in Chapter 8.

Special Topic 2.5



Using Dialog Boxes for Input and Output

Most program users find the console window rather old-fashioned. The easiest alternative is to create a separate pop-up window for each input.



An Input Dialog Box

Call the static `showInputDialog` method of the `JOptionPane` class, and supply the string that prompts the input from the user. For example,

```
String input = JOptionPane.showInputDialog("Enter price:");
```

That method returns a `String` object. Of course, often you need the input as a number. Use the `Integer.parseInt` and `Double.parseDouble` methods to convert the string to a number:

```
double price = Double.parseDouble(input);
```

You can also display output in a dialog box:

```
JOptionPane.showMessageDialog(null, "Price: " + price);
```

FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a complete program that uses option panes for input and output.



VIDEO EXAMPLE 2.2

Computing Distances on Earth

Learn how to write a program that computes the distance between any two points on Earth. Go to wiley.com/go/bj102videos to view Video Example 2.2.



© jamysavv/Stockphoto.



Computing & Society 2.2 International Alphabets and Unicode

The English alphabet is pretty simple: upper- and lowercase *a* to *z*. Other European languages have accent marks and special characters. For example, German has three so-called *umlaut* characters, ä, ö, ü, and a *double-s* character ß. These are not optional frills; you couldn't write a page of German text without using these characters a few times. German keyboards have keys for these characters.



The German Keyboard Layout

Many countries don't use the Roman script at all. Russian, Greek, Hebrew, Arabic, and Thai letters, to name just a

few, have completely different shapes. To complicate matters, Hebrew and Arabic are typed from right to left. Each of these alphabets has about as many characters as the English alphabet.



called **Unicode** that is capable of encoding text in essentially all written languages of the world. An early version of Unicode used 16 bits for each character. The Java `char` type corresponds to that encoding.

Today Unicode has grown to a 21-bit code, with definitions for over 100,000 characters. There are even plans to add codes for extinct languages, such as Egyptian hieroglyphics. Unfortunately, that means that a Java `char` value does not always correspond to a Unicode character. Some characters in languages such as Chinese or ancient Egyptian occupy two `char` values.



The Chinese Script

CHAPTER SUMMARY

Declare variables with appropriate names and types.

- A variable is a storage location with a name.
- When declaring a variable, you usually specify an initial value.
- When declaring a variable, you also specify the type of its values.
- Use the `int` type for numbers that cannot have a fractional part.
- Use the `double` type for floating-point numbers.
- By convention, variable names should start with a lowercase letter.
- An assignment statement stores a new value in a variable, replacing the previously stored value.
- The assignment operator = does *not* denote mathematical equality.



- You cannot change the value of a variable that is defined as `final`.
- Use comments to add explanations for humans who read your code. The compiler ignores comments.



Write arithmetic expressions in Java.



- Mixing integers and floating-point values in an arithmetic expression yields a floating-point value.
- The `++` operator adds 1 to a variable; the `--` operator subtracts 1.
- If both arguments of `/` are integers, the remainder is discarded.
- The `%` operator computes the remainder of an integer division.
- The Java library declares many mathematical functions, such as `Math.sqrt` (square root) and `Math.pow` (raising to a power).
- You use a cast (`typeName`) to convert a value to a different type.

Write programs that read user input and print formatted output.



- Java classes are grouped into packages. Use the `import` statement to use classes from packages.
- Use the `Scanner` class to read keyboard input in a console window.
- Use the `printf` method to specify how values should be formatted.
- The API (Application Programming Interface) documentation lists the classes and methods of the Java library.

	Month	Day	Year	Amount
January	14	120	15	Jan 14
February	29	120	15	Feb 29
March	14	120	15	Mar 14
April	15	120	15	Apr 15
May	7	120	15	May 7
June	20	120	15	Jun 20
July	15	120	15	Jul 15
August	14	120	15	Aug 14
September	3	120	15	Sep 3
October	22	120	15	Oct 22
November	1	120	15	Nov 1
December	22	120	15	Dec 22

Carry out hand calculations when developing an algorithm.

- Pick concrete values for a typical situation to use in a hand calculation.

Write programs that process strings.



- Strings are sequences of characters.
- The `length` method yields the number of characters in a string.
- Use the `+` operator to *concatenate* strings; that is, to put them together to yield a longer string.
- Whenever one of the arguments of the `+` operator is a string, the other argument is converted to a string.
- Use the `next` method of the `Scanner` class to read a string containing a single word.
- String positions are counted starting with 0.
- Use the `substring` method to extract a part of a string.

W O R D



STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.io.PrintStream	log10	java.lang.System	java.util.Scanner
printf	max	in	next
java.lang.Double	min	java.math.BigDecimal	nextDouble
parseDouble	pow	add	nextInt
java.lang.Integer	round	multiply	javax.swing.JOptionPane
MAX_VALUE	sin	subtract	showInputDialog
MIN_VALUE	sqrt	java.math.BigInteger	showMessageDialog
parseInt	tan	add	
java.lang.Math	toDegrees	multiply	
PI	toRadians	subtract	
abs	java.lang.String		
cos	charAt		
exp	length		
floorMod	substring		
log			

REVIEW EXERCISES

■ R2.1 Write declarations for storing the following quantities. Choose between integers and floating-point numbers. Declare constants when appropriate.

- a. The number of days per week
- b. The number of days until the end of the semester
- c. The number of centimeters in an inch
- d. The height of the tallest person in your class, in centimeters

■ R2.2 What is the value of `mystery` after this sequence of statements?

```
int mystery = 1;
mystery = 1 - 2 * mystery;
mystery = mystery + 1;
```

■ R2.3 What is wrong with the following sequence of statements?

```
int mystery = 1;
mystery = mystery + 1;
int mystery = 1 - 2 * mystery;
```

■■ R2.4 Write the following mathematical expressions in Java.

$$s = s_0 + v_0 t + \frac{1}{2} g t^2$$

$$FV = PV \cdot \left(1 + \frac{\text{INT}}{100}\right)^{\text{YRS}}$$

$$G = 4\pi^2 \frac{a^3}{p^2(m_1 + m_2)}$$

$$c = \sqrt{a^2 + b^2 - 2ab \cos \gamma}$$

■■ R2.5 Write the following Java expressions in mathematical notation.

- a. `dm = m * (Math.sqrt(1 + v / c) / Math.sqrt(1 - v / c) - 1);`
- b. `volume = Math.PI * r * r * h;`
- c. `volume = 4 * Math.PI * Math.pow(r, 3) / 3;`
- d. `z = Math.sqrt(x * x + y * y);`

- ■ R2.6 What are the values of the following expressions? In each line, assume that

```
double x = 2.5;
double y = -1.5;
int m = 18;
int n = 4;

a. x + n * y - (x + n) * y
b. m / n + m % n
c. 5 * x - n / 5
d. 1 - (1 - (1 - (1 - n))))
e. Math.sqrt(Math.sqrt(n))
```

- ■ R2.7 What are the values of the following expressions, assuming that *n* and *m* have type *int*, *n* is 17, and *m* is 18?

```
a. n / 10 + n % 10
b. n % 2 + m % 2
c. (m + n) / 2
d. (m + n) / 2.0
e. (int) (0.5 * (m + n))
f. (int) Math.round(0.5 * (m + n))
```

- ■ R2.8 What are the values of the following expressions? In each line, assume that

```
String s = "Hello";
String t = "World";

a. s.length() + t.length()
b. s.substring(1, 2)
c. s.substring(s.length() / 2, s.length())
d. s + t
e. t + s
```

- ■ R2.9 Assuming that *a* and *b* are variables of type *int*, fill in the following table:

<i>a</i>	<i>b</i>	<code>Math.pow(a, b)</code>	<code>Math.max(a, b)</code>	<i>a</i> / <i>b</i>	<i>a</i> % <i>b</i>	<code>Math.floorMod(a, b)</code>
2	3					
3	2					
2	-3					
3	-2					
-3	2					
-3	-2					

- ■ R2.10 Suppose *direction* is an integer angle between 0 and 359 degrees. You turn by a given angle and update the direction as

```
direction = (direction + turn) % 360;
```

In which situation do you get the wrong result? How can you fix that without using the `Math.floorMod` method described in Java 8 Note 2.1?

- R2.11 Find at least five *compile-time* errors in the following program.

```
public class HasErrors
{
    public static void main()
    {
        System.out.print(Please enter two numbers:)
        x = in.readDouble;
        y = in.readDouble;
        System.out.println("The sum is " + x + y);
    }
}
```

- R2.12 Find three *run-time* errors in the following program.

```
public class HasErrors
{
    public static void main(String[] args)
    {
        int x = 0;
        int y = 0;
        Scanner in = new Scanner("System.in");
        System.out.print("Please enter an integer:");
        x = in.readInt();
        System.out.print("Please enter another integer: ");
        x = in.readInt();
        System.out.println("The sum is " + x + y);
    }
}
```

- R2.13 Consider the following code segment.

```
double purchase = 19.93;
double payment = 20.00;
double change = payment - purchase;
System.out.println(change);
```

The code segment prints the change as 0.0700000000000028. Explain why. Give a recommendation to improve the code so that users will not be confused.

- R2.14 Explain the differences between 2, 2.0, '2', "2", and "2.0".

- R2.15 Explain what each of the following program segments computes.

- a. `x = 2;`
`y = x + x;`
- b. `s = "2";`
`t = s + s;`

- R2.16 Write pseudocode for a program that reads a word and then prints the first character, the last character, and the characters in the middle. For example, if the input is Harry, the program prints H y arr.

- R2.17 Write pseudocode for a program that reads a name (such as Harold James Morgan) and then prints a monogram consisting of the initial letters of the first, middle, and last name (such as HJM).

- R2.18 Write pseudocode for a program that computes the first and last digit of a number. For example, if the input is 23456, the program should print 2 and 6.
Hint: %, Math.log10.

■ **R2.19** Modify the pseudocode for the program in How To 2.1 so that the program gives change in quarters, dimes, and nickels. You can assume that the price is a multiple of 5 cents. To develop your pseudocode, first work with a couple of specific values.

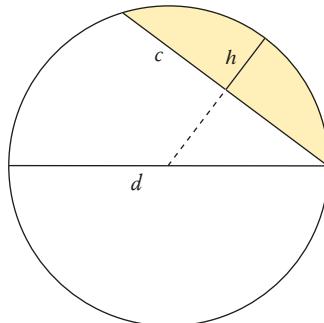
■■ **R2.20** A cocktail shaker is composed of three cone sections.

Using realistic values for the radii and heights, compute the total volume, using the formula given in Self Check 25 for a cone section. Then develop an algorithm that works for arbitrary dimensions.



© Media Bakery.

■■■ **R2.21** You are cutting off a piece of pie like this, where c is the length of the straight part (called the chord length) and h is the height of the piece.



There is an approximate formula for the area: $A \approx \frac{2}{3}ch + \frac{h^3}{2c}$

However, h is not so easy to measure, whereas the diameter d of a pie is usually well-known. Calculate the area where the diameter of the pie is 12 inches and the chord length of the segment is 10 inches. Generalize to an algorithm that yields the area for any diameter and chord length.

■■ **R2.22** The following pseudocode describes how to obtain the name of a day, given the day number (0 = Sunday, 1 = Monday, and so on.)

Declare a string called names containing "SunMonTueWedThuFriSat".

Compute the starting position as 3 x the day number.

Extract the substring of names at the starting position with length 3.

Check this pseudocode, using the day number 4. Draw a diagram of the string that is being computed, similar to Figure 5.

■■■ **R2.23** Suppose you are given a string **str** and two positions **i** and **j**, where **i** comes before **j**. The following pseudocode describes how to swap two letters in a word.

We are given a string str and two positions i and j. (i comes before j)

Set first to the substring from the start of the string to the last position before i.

Set middle to the substring from positions i + 1 to j - 1.

Set last to the substring from position j + 1 to the end of the string.

Concatenate the following five strings: first, the string containing just the character at position j, middle, the string containing just the character at position i, and last.

Check this pseudocode, using the string "Gateway" and positions 2 and 4. Draw a diagram of the string that is being computed, similar to Figure 5.

■■ R2.24 How do you get the first character of a string? The last character? How do you remove the first character? The last character?

■■ R2.25 For each of the following computations in Java, determine whether the result is exact, an overflow, or a roundoff error.

- a. $2.0 - 1.1$
- b. $1.0E6 * 1.0E6$
- c. $65536 * 65536$
- d. $1_000_000L * 1_000_000L$

■■■ R2.26 Write a program that prints the values

$$\begin{aligned} 3 * 1000 * 1000 * 1000 \\ 3.0 * 1000 * 1000 * 1000 \end{aligned}$$

Explain the results.

■ R2.27 This chapter contains a number of recommendations regarding variables and constants that make programs easier to read and maintain. Briefly summarize these recommendations.

PRACTICE EXERCISES

■ E2.1 Write a program that displays the dimensions of a letter-size (8.5×11 inches) sheet of paper in millimeters. There are 25.4 millimeters per inch. Use constants and comments in your program.

■ E2.2 Write a program that computes and displays the perimeter of a letter-size (8.5×11 inches) sheet of paper and the length of its diagonal.

■ E2.3 Write a program that reads a number and displays the square, cube, and fourth power. Use the `Math.pow` method only for the fourth power.

■■ E2.4 Write a program that prompts the user for two integers and then prints

- The sum
- The difference
- The product
- The average
- The distance (absolute value of the difference)
- The maximum (the larger of the two)
- The minimum (the smaller of the two)

Hint: The `max` and `min` functions are declared in the `Math` class.

■■ E2.5 Enhance the output of Exercise E2.4 so that the numbers are properly aligned:

Sum:	45
Difference:	-5
Product:	500
Average:	22.50
Distance:	5
Maximum:	25
Minimum:	20

- **E2.6** Write a program that prompts the user for a measurement in meters and then converts it to miles, feet, and inches.
- **E2.7** Write a program that prompts the user for a radius and then prints
 - The area and circumference of a circle with that radius
 - The volume and surface area of a sphere with that radius
- **E2.8** Write a program that asks the user for the lengths of the sides of a rectangle. Then print
 - The area and perimeter of the rectangle
 - The length of the diagonal (use the Pythagorean theorem)
- **E2.9** Improve the program discussed in How To 2.1 to allow input of quarters in addition to bills.
- **E2.10** Write a program that helps a person decide whether to buy a hybrid car. Your program's inputs should be:
 - The cost of a new car
 - The estimated miles driven per year
 - The estimated gas price
 - The efficiency in miles per gallon
 - The estimated resale value after 5 years

Compute the total cost of owning the car for five years. (For simplicity, we will not take the cost of financing into account.) Obtain realistic prices for a new and used hybrid and a comparable car from the Web. Run your program twice, using today's gas price and 15,000 miles per year. Include pseudocode and the program runs with your assignment.



© assecit/iStockphoto.

- **E2.11** Write a program that asks the user to input
 - The number of gallons of gas in the tank
 - The fuel efficiency in miles per gallon
 - The price of gas per gallon
- Then print the cost per 100 miles and how far the car can go with the gas in the tank.
- **E2.12** *File names and extensions.* Write a program that prompts the user for the drive letter (C), the path (\Windows\System), the file name (Readme), and the extension (txt). Then print the complete file name C:\Windows\System\Readme.txt. (If you use UNIX or a Macintosh, skip the drive name and use / instead of \ to separate directories.)
- **E2.13** Write a program that reads a number between 1,000 and 999,999 from the user, where the user enters a comma in the input. Then print the number without a comma. Here is a sample dialog; the user input is in color:

```
Please enter an integer between 1,000 and 999,999: 23,456
23456
```

Hint: Read the input as a string. Measure the length of the string. Suppose it contains n characters. Then extract substrings consisting of the first $n - 4$ characters and the last three characters.

- E2.14** Write a program that reads a number between 1,000 and 999,999 from the user and prints it with a comma separating the thousands. Here is a sample dialog; the user input is in color:

```
Please enter an integer between 1000 and 999999: 23456
23,456
```

- E2.15** *Printing a grid.* Write a program that prints the following grid to play tic-tac-toe.

```
+---+---+
| | | |
+---+---+
| | | |
+---+---+
| | | |
+---+---+
```

Of course, you could simply write seven statements of the form

```
System.out.println("+-+-+-+");
```

You should do it the smart way, though. Declare string variables to hold two kinds of patterns: a comb-shaped pattern and the bottom line. Print the comb three times and the bottom line once.

- E2.16** Write a program that reads in an integer and breaks it into a sequence of individual digits. For example, the input 16384 is displayed as

```
1 6 3 8 4
```

You may assume that the input has no more than five digits and is not negative.

- E2.17** Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times. Here is a sample run. User input is in color.

```
Please enter the first time: 0900
Please enter the second time: 1730
8 hours 30 minutes
```

Extra credit if you can deal with the case where the first time is later than the second:

```
Please enter the first time: 1730
Please enter the second time: 0900
15 hours 30 minutes
```

- E2.18** *Writing large letters.* A large letter H can be produced like this:

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

It can be declared as a string literal like this:

```
final String LETTER_H = "*   *\n*   *\n*****\n*   *\n*   *";
```

Print the string with `System.out.printf`. The `\n` format specifiers cause line breaks in the output. Do the same for the letters E, L, and O. Then write in large letters:

```
H
E
L
L
O
```

- E2.19** Write a program that transforms numbers 1, 2, 3, ..., 12 into the corresponding month names January, February, March, ..., December. *Hint:* Make a very long string "January February March ...", in which you add spaces such that each month name has *the same length*. Then use substring to extract the month you want.
- E2.20** Write a program that prints a Christmas tree, as shown below. Remember to use escape sequences.



© José Luis Gutiérrez/iStockphoto

PROGRAMMING PROJECTS

- P2.1** Easter Sunday is the first Sunday after the first full moon of spring. To compute the date, you can use this algorithm, invented by the mathematician Carl Friedrich Gauss in 1800:

1. Let y be the year (such as 1800 or 2001).
2. Divide y by 19 and call the remainder a . Ignore the quotient.
3. Divide y by 100 to get a quotient b and a remainder c .
4. Divide b by 4 to get a quotient d and a remainder e .
5. Divide $8 * b + 13$ by 25 to get a quotient g . Ignore the remainder.
6. Divide $19 * a + b - d - g + 15$ by 30 to get a remainder h . Ignore the quotient.
7. Divide c by 4 to get a quotient j and a remainder k .
8. Divide $a + 11 * h$ by 319 to get a quotient m . Ignore the remainder.
9. Divide $2 * e + 2 * j - k - h + m + 32$ by 7 to get a remainder r . Ignore the quotient.
10. Divide $h - m + r + 90$ by 25 to get a quotient n . Ignore the remainder.
11. Divide $h - m + r + n + 19$ by 32 to get a remainder p . Ignore the quotient.

Then Easter falls on day p of month n . For example, if y is 2001:

$$\begin{array}{llll} a = 6 & g = 6 & m = 0 & n = 4 \\ b = 20, \quad c = 1 & h = 18 & r = 6 & p = 15 \\ d = 5, \quad e = 0 & j = 0, \quad k = 1 & & \end{array}$$

Therefore, in 2001, Easter Sunday fell on April 15. Write a program that prompts the user for a year and prints out the month and day of Easter Sunday.

- P2.2** In this project, you will perform calculations with triangles. A triangle is defined by the x - and y -coordinates of its three corner points.

Your job is to compute the following properties of a given triangle:

- the lengths of all sides
- the angles at all corners
- the perimeter
- the area

Supply a program that prompts a user for the corner point coordinates and produces a nicely formatted table of the triangle properties.

- **Business P2.3** The following pseudocode describes how a bookstore computes the price of an order from the total price and the number of the books that were ordered.

Read the total book price and the number of books.
Compute the tax (7.5 percent of the total book price).
Compute the shipping charge (\$2 per book).
The price of the order is the sum of the total book price, the tax, and the shipping charge.
Print the price of the order.

Translate this pseudocode into a Java program.

- **Business P2.4** The following pseudocode describes how to turn a string containing a ten-digit phone number (such as "4155551212") into a more readable string with parentheses and dashes, like this: "(415) 555-1212".

Take the substring consisting of the first three characters and surround it with "(" and ")". This is the area code.
Concatenate the area code, the substring consisting of the next three characters, a hyphen, and the substring consisting of the last four characters. This is the formatted number.

Translate this pseudocode into a Java program that reads a telephone number into a string variable, computes the formatted number, and prints it.

- **Business P2.5** The following pseudocode describes how to extract the dollars and cents from a price given as a floating-point value. For example, a price 2.95 yields values 2 and 95 for the dollars and cents.

Assign the price to an integer variable dollars.
Multiply the difference price - dollars by 100 and add 0.5.
Assign the result to an integer variable cents.

Translate this pseudocode into a Java program. Read a price and print the dollars and cents. Test your program with inputs 2.95 and 4.35.

- **Business P2.6** *Giving change.* Implement a program that directs a cashier how to give change. The program has two inputs: the amount due and the amount received from the customer. Display the dollars, quarters, dimes, nickels, and pennies that the customer should receive in return. In order to avoid roundoff errors, the program user should supply both amounts in pennies, for example 274 instead of 2.74.



© Captainflash/Stockphoto.

- **Business P2.7** An online bank wants you to create a program that shows prospective customers how their deposits will grow. Your program should read the initial balance and the annual interest rate. Interest is compounded monthly. Print out the balances after the first three months. Here is a sample run:

```
Initial balance: 1000
Annual interest rate in percent: 6.0
After first month:    1005.00
After second month:   1010.03
After third month:    1015.08
```

- **Business P2.8** A video club wants to reward its best members with a discount based on the member's number of movie rentals and the number of new members referred by the

member. The discount is in percent and is equal to the sum of the rentals and the referrals, but it cannot exceed 75 percent. (*Hint:* Math.min.) Write a program `DiscountCalculator` to calculate the value of the discount.

Here is a sample run:

```
Enter the number of movie rentals: 56
Enter the number of members referred to the video club: 3
The discount is equal to: 59.00 percent.
```

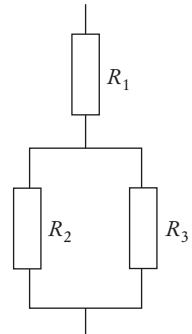
■ Science P2.9 Consider the circuit at right.

Write a program that reads the resistances of the three resistors and computes the total resistance, using Ohm's law.

■■ Science P2.10 The dew point temperature T_d can be calculated (approximately) from the relative humidity RH and the actual temperature T by

$$T_d = \frac{b \cdot f(T, RH)}{a - f(T, RH)}$$

$$f(T, RH) = \frac{a \cdot T}{b + T} + \ln(RH)$$



where $a = 17.27$ and $b = 237.7^\circ\text{C}$.

Write a program that reads the relative humidity (between 0 and 1) and the temperature (in degrees C) and prints the dew point value. Use the Java method `Math.log` to compute the natural logarithm.

■■■ Science P2.11 The pipe clip temperature sensors shown here are robust sensors that can be clipped directly onto copper pipes to measure the temperature of the liquids in the pipes.



Each sensor contains a device called a *thermistor*. Thermistors are semiconductor devices that exhibit a temperature-dependent resistance described by:

$$R = R_0 e^{\beta \left(\frac{1}{T} - \frac{1}{T_0} \right)}$$

where R is the resistance (in Ω) at the temperature T (in $^\circ\text{K}$), and R_0 is the resistance (in Ω) at the temperature T_0 (in $^\circ\text{K}$). β is a constant that depends on the material used to make the thermistor. Thermistors are specified by providing values for R_0 , T_0 , and β .

The thermistors used to make the pipe clip temperature sensors have $R_0 = 1075 \Omega$ at $T_0 = 85^\circ\text{C}$, and $\beta = 3969 \text{ }^\circ\text{K}$. (Notice that β has units of $^\circ\text{K}$. Recall that the temperature in $^\circ\text{K}$ is obtained by adding 273 to the temperature in $^\circ\text{C}$.) The liquid temperature, in $^\circ\text{C}$, is determined from the resistance R , in Ω , using

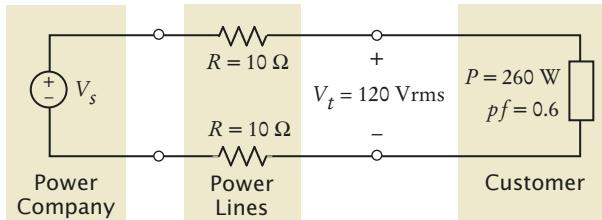
$$T = \frac{\beta T_0}{T_0 \ln\left(\frac{R}{R_0}\right) + \beta} - 273$$

Write a Java program that prompts the user for the thermistor resistance R and prints a message giving the liquid temperature in °C.

- Science P2.12** The circuit shown below illustrates some important aspects of the connection between a power company and one of its customers. The customer is represented by three parameters, V_t , P , and pf . V_t is the voltage accessed by plugging into a wall outlet. Customers depend on having a dependable value of V_t in order for their appliances to work properly. Accordingly, the power company regulates the value of V_t carefully. P describes the amount of power used by the customer and is the primary factor in determining the customer's electric bill. The power factor, pf , is less familiar. (The power factor is calculated as the cosine of an angle so that its value will always be between zero and one.) In this problem you will be asked to write a Java program to investigate the significance of the power factor.



© TebNadiStockphoto.

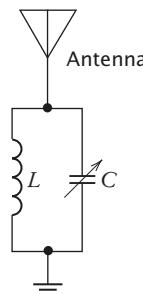


In the figure, the power lines are represented, somewhat simplistically, as resistances in Ohms. The power company is represented as an AC voltage source. The source voltage, V_s , required to provide the customer with power P at voltage V_t can be determined using the formula

$$V_s = \sqrt{\left(V_t + \frac{2RP}{V_t}\right)^2 + \left(\frac{2RP}{pfV_t}\right)^2 (1 - pf^2)}$$

(V_s has units of Vrms.) This formula indicates that the value of V_s depends on the value of pf . Write a Java program that prompts the user for a power factor value and then prints a message giving the corresponding value of V_s , using the values for P , R , and V_t shown in the figure above.

- Science P2.13** Consider the following tuning circuit connected to an antenna, where C is a variable capacitor whose capacitance ranges from C_{\min} to C_{\max} .



The tuning circuit selects the frequency $f = \frac{2\pi}{\sqrt{LC}}$. To design this circuit for a given frequency, take $C = \sqrt{C_{\min} C_{\max}}$ and calculate the required inductance L from f and C . Now the circuit can be tuned to any frequency in the range $f_{\min} = \frac{2\pi}{\sqrt{LC_{\max}}}$ to $f_{\max} = \frac{2\pi}{\sqrt{LC_{\min}}}$.

Write a Java program to design a tuning circuit for a given frequency, using a variable capacitor with given values for C_{\min} and C_{\max} . (A typical input is $f = 16.7$ MHz, $C_{\min} = 14$ pF, and $C_{\max} = 365$ pF.) The program should read in f (in Hz), C_{\min} and C_{\max} (in F), and print the required inductance value and the range of frequencies to which the circuit can be tuned by varying the capacitance.

- **Science P2.14** According to the Coulomb force law, the electric force between two charged particles of charge Q_1 and Q_2 Coulombs, that are a distance r meters apart, is

$$F = \frac{Q_1 Q_2}{4\pi\epsilon r^2} \text{ Newtons, where } \epsilon = 8.854 \times 10^{-12} \text{ Farads/meter. Write a program}$$

that calculates the force on a pair of charged particles, based on the user input of Q_1 Coulombs, Q_2 Coulombs, and r meters, and then computes and displays the electric force.

ANSWERS TO SELF-CHECK QUESTIONS

- 1.** One possible answer is

```
int bottlesPerCase = 8;
```

You may choose a different variable name or a different initialization value, but your variable should have type int.

- 2.** There are three errors:

- You cannot have spaces in variable names.
- The variable type should be double because it holds a fractional value.
- There is a semicolon missing at the end of the statement.

- 3.** `double unitPrice = 1.95;`
`int quantity = 2;`

- 4.** `System.out.print("Total price: ");`
`System.out.println(unitPrice * quantity);`

- 5.** Change the declaration of cansPerPack to
`int cansPerPack = 4;`

- 6.** You need to use a */ delimiter to close a comment that begins with a /*:

```
double canVolume = 0.355;  

/* Liters in a 12-ounce can */
```

- 7.** The program would compile, and it would display the same result. However, a person reading the program might find it confusing that fractional cans are being considered.

- 8.** Its value is modified by the assignment statement.

- 9.** Assignment would occur when one car is replaced by another in the parking space.

- 10.** `double interest = balance * percent / 100;`

- 11.** `double sideLength = Math.sqrt(area);`

- 12.** `4 * Math.PI * Math.pow(radius, 3) / 3`
or `(4.0 / 3) * Math.PI * Math.pow(radius, 3)`,
but not `(4 / 3) * Math.PI * Math.pow(radius, 3)`

- 13.** 172 and 9

- 14.** It is the second-to-last digit of n. For example, if n is 1729, then n / 10 is 172, and (n / 10) % 10 is 2.

- 15.** `System.out.print("How old are you? ");`
`int age = in.nextInt();`

- 16.** There is no prompt that alerts the program user to enter the quantity.

17. The second statement calls `nextInt`, not `nextDouble`. If the user were to enter a price such as 1.95, the program would be terminated with an “input mismatch exception”.

18. There is no colon and space at the end of the prompt. A dialog would look like this:

Please enter the number of cans6

19. The total volume is 10

There are four spaces between `is` and `10`. One space originates from the format string (the space between `s` and `%`), and three spaces are added before `10` to achieve a field width of 5.

20. Here is a simple solution:

```
System.out.printf("Bottles: %d%n", bottles);
System.out.printf("Cans:    %d%n", cans);
```

Note the spaces after `Cans:`. Alternatively, you can use format specifiers for the strings. You can even combine all output into a single statement:

```
System.out.printf("%-9s%8d%-9s%8d%n",
"Bottles: ", bottles, "Cans:", cans);
```

21.

```
int pairs = (totalWidth - tileSize)
           / (2 * tileSize);
int tiles = 1 + 2 * pairs;
double gap = (totalWidth -
              tiles * tileSize) / 2.0;
```

Be sure that `pairs` is declared as an `int`.

22. Now there are groups of four tiles (gray/white/gray/black) following the initial black tile. Therefore, the algorithm is now

```
number of groups = integer part of (total width - tile width) /
                  (4 x tile width)
number of tiles = 1 + 4 x number of groups
```

The formula for the gap is not changed.

23. Clearly, the answer depends only on whether the row and column numbers are even or odd, so let's first take the remainder after dividing by 2. Then we can enumerate all expected answers:

Row % 2	Column % 2	Color
0	0	0
0	1	1
1	0	1
1	1	0

In the first three entries of the table, the color is simply the sum of the remainders. In the

fourth entry, the sum would be 2, but we want a zero. We can achieve that by taking another remainder operation:

`color = ((row % 2) + (column % 2)) % 2`

24. In nine years, the repair costs increased by \$1,400. Therefore, the increase per year is $\$1,400 / 9 \approx \156 . The repair cost in year 3 would be $\$100 + 2 \times \$156 = \$412$. The repair cost in year `n` is $\$100 + n \times \156 . To avoid accumulation of roundoff errors, it is actually a good idea to use the original expression that yielded \$156, that is,

`Repair cost in year n = 100 + n x 1400 / 9`

25. The pseudocode follows easily from the equations:

$$\text{bottom volume} = \pi \times r_1^2 \times h_1$$

$$\text{top volume} = \pi \times r_2^2 \times h_2$$

$$\text{middle volume} = \pi \times (r_1^2 + r_1 \times r_2 + r_2^2) \times h_3 / 3$$

$$\text{total volume} = \text{bottom volume} + \text{top volume} + \text{middle volume}$$

Measuring a typical wine bottle yields

$$r_1 = 3.6, r_2 = 1.2, h_1 = 15, h_2 = 7, h_3 = 6$$

(all in centimeters). Therefore,

$$\text{bottom volume} = 610.73$$

$$\text{top volume} = 31.67$$

$$\text{middle volume} = 135.72$$

$$\text{total volume} = 778.12$$

The actual volume is 750 ml, which is close enough to our computation to give confidence that it is correct.

26. The length is 12. The space counts as a character.

27. `str.substring(8, 12)` or `str.substring(8)`

28. `str = str + "ming";`

29. Hy

30.

```
String first = in.next();
String middle = in.next();
String last = in.next();
```

WORKED EXAMPLE 2.1

Computing the Cost of Stamps

You are asked to simulate a postage stamp vending machine. A customer inserts dollar bills into the vending machine and then pushes a “purchase” button. The vending machine gives out as many first-class stamps as the customer paid for, and returns the change in penny (one-cent) stamps. A first-class stamp cost 47 cents at the time this book was written.

Step 1 Understand the problem: What are the inputs? What are the desired outputs?

In this problem, there is one input:

- The amount of money the customer inserts

There are two desired outputs:

- The number of first-class stamps the machine returns
- The number of penny stamps the machine returns

Step 2 Work out examples by hand.

Let’s assume that a first-class stamp costs 47 cents and the customer inserts \$1.00. That’s enough for two stamps (94 cents) but not enough for three stamps (\$1.41). Therefore, the machine returns two first-class stamps and 6 penny stamps.

Step 3 Write pseudocode for computing the answers.

Given an amount of money and the price of a first-class stamp, how can you compute how many first-class stamps can be purchased with the money? Clearly, the answer is related to the quotient

$$\frac{\text{amount of money}}{\text{price of first-class stamp}}$$

For example, suppose the customer paid \$1.00. Use a pocket calculator to compute the quotient: $\$1.00/\$0.47 = 2.13$.

How do you get “2 stamps” out of 2.13? It’s the quotient without the remainder. In Java, this is easy to compute if both arguments are integers. Therefore, let’s switch our computation to pennies. Then we have

$$\text{number of first-class stamps} = 100 / 47 \text{ (integer division, without remainder)}$$

What if the user inputs two dollars? Then the numerator becomes 200. What if the price of a stamp goes up? A more general equation is

$$\text{number of first-class stamps} = 100 \times \text{dollars} / \text{price of first-class stamps in cents}$$

How about the change? Here is one way of computing it. When the customer gets the stamps, the change is the customer payment, reduced by the value of the stamps purchased. In our example, the change is 6 cents—the difference between 100 and $2 \cdot 47$. Here is the general formula:

$$\text{change} = 100 \times \text{dollars} - \text{number of first-class stamps} \times \text{price of first-class stamp}$$

Step 4 Declare the variables and constants that you need, and specify their types.

Here, we have three variables:

- dollars
- firstClassStamps
- change

There is one constant, FIRST_CLASS_STAMP_PRICE.

WE2 Chapter 2 Fundamental Data Types

The variable `dollars` and constant `FIRST_CLASS_STAMP_PRICE` must be of type `int` because the computation of `firstClassStamps` uses integer division. The remaining variables are also integers, counting the number of first-class and penny stamps. Thus, we have

```
final int FIRST_CLASS_STAMP_PRICE = 47; // Price in pennies
int dollars;
int firstClassStamps;
int change;
```

Step 5 Turn the pseudocode into Java statements.

Our computation depends on the number of dollars that the user provides. Translating the math into Java yields the following statements:

```
int firstClassStamps = 100 * dollars / FIRST_CLASS_STAMP_PRICE;
int change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
```

Step 6 Provide input and output.

Before starting the computation, we prompt the user for the number of dollars:

```
System.out.print("Enter number of dollars: ");
int dollars = in.nextInt();
```

When the computation is finished, we display the result.

```
System.out.printf("First class stamps: %d\n", firstClassStamps);
System.out.printf("Penny stamps: %d\n", change);
```

Step 7 Provide a class with a `main` method.

Here is the complete program.

worked_example_1/StampMachine.java

```
1 import java.util.Scanner;
2 /**
3  * This program simulates a stamp machine that receives dollar bills and dispenses
4  * first class and penny stamps.
5 */
6 public class StampMachine
7 {
8     public static void main(String[] args)
9     {
10        Scanner in = new Scanner(System.in);
11
12        final int FIRST_CLASS_STAMP_PRICE = 47; // Price in pennies
13
14        System.out.print("Enter number of dollars: ");
15        int dollars = in.nextInt();
16
17        // Compute and print the number of stamps to dispense
18
19        int firstClassStamps = 100 * dollars / FIRST_CLASS_STAMP_PRICE;
20        int change = 100 * dollars - firstClassStamps * FIRST_CLASS_STAMP_PRICE;
21        System.out.printf("First class stamps: %d\n", firstClassStamps);
22        System.out.printf("Penny stamps: %d\n", change);
23    }
24 }
```

Program Run

```
Enter number of dollars: 2
First class stamps: 4
Penny stamps: 12
```

WORKED EXAMPLE 2.2

Computing Travel Time



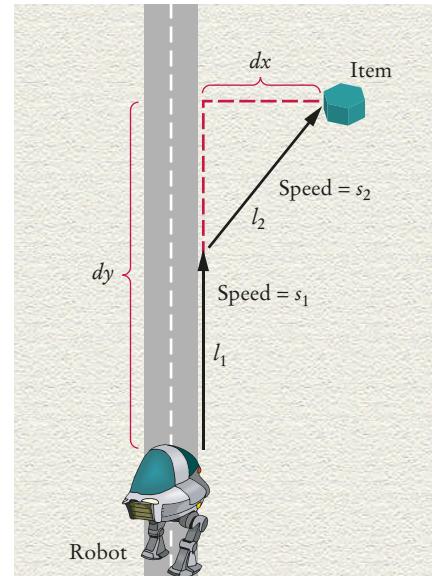
In this Worked Example, we develop a hand calculation to compute the time that a robot requires to retrieve an item from rocky terrain.



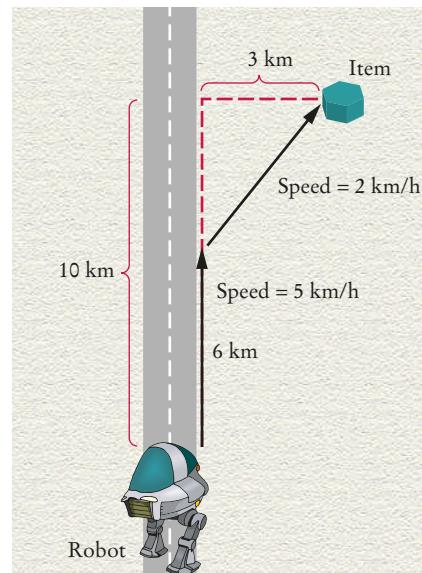
Problem Statement A robot needs to retrieve an item that is located in rocky terrain adjacent to a road. The robot can travel at a faster speed on the road than on the rocky terrain, so it will want to do so for a certain distance before moving on a straight line to the item.

Your task is to compute the total time taken by the robot to reach its goal, given the following inputs:

- The distance between the robot and the item in the x - and y -direction (dx and dy)
- The speed of the robot on the road and the rocky terrain (s_1 and s_2)
- The length l_1 of the first segment (on the road)



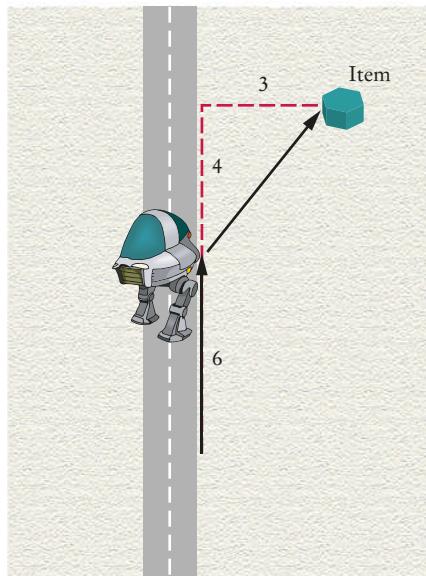
To make the problem more concrete, let's assume the following dimensions:



The total time is the time for traversing both segments. The time to traverse the first segment is simply the length of the segment divided by the speed: 6 km divided by 5 km/h, or 1.2 hours.

WE4 Chapter 2 Fundamental Data Types

To compute the time for the second segment, we first need to know its length. It is the hypotenuse of a right triangle with side lengths 3 and 4.



Therefore, its length is $\sqrt{3^2 + 4^2} = 5$. At 2 km/h, it takes 2.5 hours to traverse it. That makes the total travel time 3.7 hours.

This computation gives us enough information to devise an algorithm for the total travel time with arbitrary arguments:

```
Time for segment 1 = l1 / s1
Length of segment 2 = square root of dx2 + (dy - l1)2
Time for segment 2 = length of segment 2 / s2
Total time = time for segment 1 + time for segment 2
```

Translated into Java, the computations are

```
double segment1Time = segment1Length / segment1Speed;
double segment2Length = Math.sqrt(Math.pow(xDistance, 2)
    + Math.pow(yDistance - segment1Length, 2));
double segment2Time = segment2Length / segment2Speed;
double totalTime = segment1Time + segment2Time;
```

Note that we use variable names that are longer and more descriptive than dx or s_1 . When you do hand calculations, it is convenient to use the shorter names, but you should change them to descriptive names in your program.

DECISIONS

CHAPTER GOALS

To implement decisions using if statements



© zennie/iStockphoto.

To compare integers, floating-point numbers, and strings

To write statements using the Boolean data type

To develop strategies for testing your programs

To validate user input

CHAPTER CONTENTS

3.1 THE IF STATEMENT 84

- SYN** if Statement 86
- PT1** Brace Layout 88
- PT2** Always Use Braces 88
- CE1** A Semicolon After the if Condition 88
- PT3** Tabs 89
- ST1** The Conditional Operator 89
- PT4** Avoid Duplication in Branches 90

3.2 COMPARING NUMBERS AND STRINGS 90

- SYN** Comparisons 91
- CE2** Exact Comparison of Floating-Point Numbers 93
- CE3** Using == to Compare Strings 94
- ST2** Lexicographic Ordering of Strings 94
- HT1** Implementing an if Statement 95
- WE1** Extracting the Middle 96
- C&S** The Denver Airport Luggage System 97

3.3 MULTIPLE ALTERNATIVES 98

- ST3** The switch Statement 101

3.4 NESTED BRANCHES 102

- PT5** Hand-Tracing 105

- CE4** The Dangling else Problem 106

- ST4** Enumeration Types 107

- VE1** Computing the Plural of an English Word

3.5 PROBLEM SOLVING: FLOWCHARTS 107

3.6 PROBLEM SOLVING: TEST CASES 110

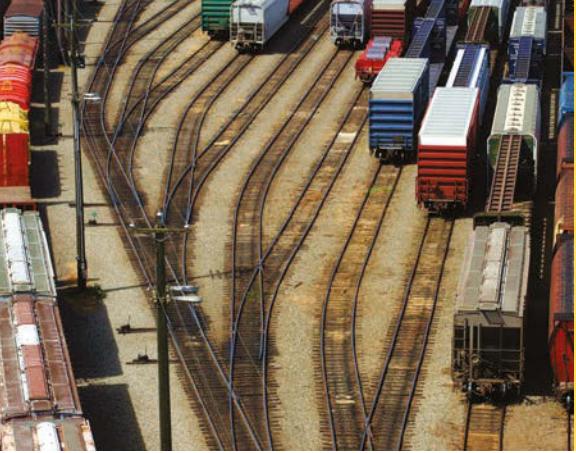
- PT6** Make a Schedule and Make Time for Unexpected Problems 111
- ST5** Logging 112

3.7 BOOLEAN VARIABLES AND OPERATORS 113

- CE5** Combining Multiple Relational Operators 115
- CE6** Confusing && and || Conditions 116
- ST6** Short-Circuit Evaluation of Boolean Operators 116
- ST7** De Morgan's Law 117

3.8 APPLICATION: INPUT VALIDATION 118

- VE2** The Genetic Code 119
- C&S** Artificial Intelligence 121



© zennie/iStockphoto.

One of the essential features of computer programs is their ability to make decisions. Like a train that changes tracks depending on how the switches are set, a program can take different actions depending on inputs and other circumstances.

In this chapter, you will learn how to program simple and complex decisions. You will apply what you learn to the task of checking user input.

3.1 The if Statement

The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

The `if` statement is used to implement a decision (see Syntax 3.1). When a condition is fulfilled, one set of statements is executed. Otherwise, another set of statements is executed.

Here is an example using the `if` statement: In many countries, the number 13 is considered unlucky. Rather than offending superstitious tenants, building owners sometimes skip the thirteenth floor; floor 12 is immediately followed by floor 14. Of course, floor 13 is not usually left empty or, as some conspiracy theorists believe, filled with secret offices and research labs. It is simply called floor 14. The computer that controls the building elevators needs to compensate for this foible and adjust all floor numbers above 13.

Let's simulate this process in Java. We will ask the user to type in the desired floor number and then compute the actual floor. When the input is above 13, then we need to decrement the input to obtain the actual floor. For example, if the user provides an input of 20, the program determines the actual floor as 19. Otherwise, we simply use the supplied floor number.

```
int actualFloor;

if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
```

The flowchart in Figure 1 shows the branching behavior.

In our example, each branch of the `if` statement contains a single statement. You can include as many statements in each branch as you like. Sometimes, it happens that



© DrGrounds/iStockphoto.

This elevator panel “skips” the thirteenth floor. The floor is not actually missing—the computer that controls the elevator adjusts the floor numbers above 13.

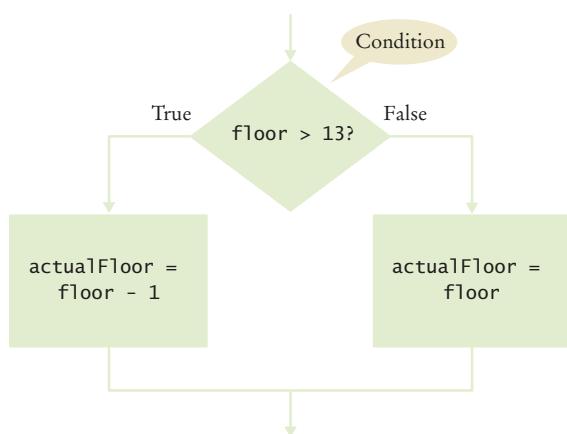


Figure 1
Flowchart for if Statement

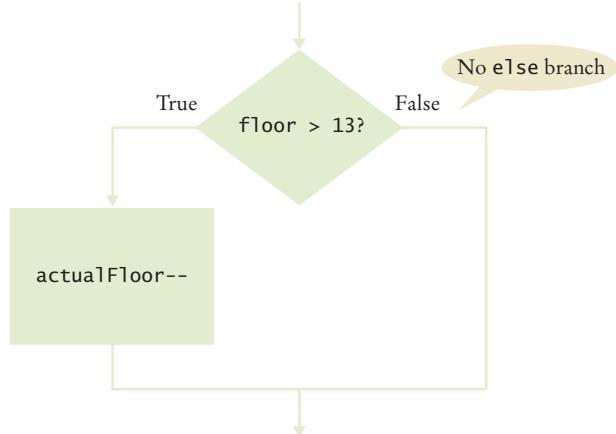


Figure 2
Flowchart for if Statement with No else Branch

there is nothing to do in the else branch of the statement. In that case, you can omit it entirely, such as in this example:

```

int actualFloor = floor;

if (floor > 13)
{
    actualFloor--;
} // No else needed
  
```

See Figure 2 for the flowchart.



© Media Bakery.

An if statement is like a fork in the road. Depending upon a decision, different parts of the program are executed.

Syntax 3.1 if Statement

```
Syntax    if (condition)
{            statements
}
```

```
if (condition) { statements1 }
else { statements2 }
```

Braces are not required if the branch contains a single statement, but it's good to always use them.

 See Programming Tip 3.2.

A condition that is true or false.
Often uses relational operators:
== != < <= > >= (See Table 1.)

 Don't put a semicolon here!
See Common Error 3.1.

Omit the else branch if there is nothing to do.

 Lining up braces is a good idea.
See Programming Tip 3.1.

```
if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
```

If the condition is true, the statement(s) in this branch are executed in sequence; if the condition is false, they are skipped.

If the condition is false, the statement(s) in this branch are executed in sequence; if the condition is true, they are skipped.

The following program puts the if statement to work. This program asks for the desired floor and then prints out the actual floor.

sec01/ElevatorSimulation.java

```

1  import java.util.Scanner;
2
3  /**
4   * This program simulates an elevator panel that skips the 13th floor.
5  */
6  public class ElevatorSimulation
7  {
8      public static void main(String[] args)
9      {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Floor: ");
12         int floor = in.nextInt();
13
14         // Adjust floor if necessary
15
16         int actualFloor;
17         if (floor > 13)
18         {
19             actualFloor = floor - 1;
20         }
21         else
22         {
23             actualFloor = floor;
24         }
25     }
26 }
```

```

24     }
25
26     System.out.println("The elevator will travel to the actual floor "
27         + actualFloor);
28 }
29 }
```

Program Run

```
Floor: 20
The elevator will travel to the actual floor 19
```



SELF CHECK

1. In some Asian countries, the number 14 is considered unlucky. Some building owners play it safe and skip *both* the thirteenth and the fourteenth floor. How would you modify the sample program to handle such a building?
2. Consider the following if statement to compute a discounted price:

```
if (originalPrice > 100)
{
    discountedPrice = originalPrice - 20;
}
else
{
    discountedPrice = originalPrice - 10;
}
```

What is the discounted price if the original price is 95? 100? 105?

3. Compare this if statement with the one in Self Check 2:

```
if (originalPrice < 100)
{
    discountedPrice = originalPrice - 10;
}
else
{
    discountedPrice = originalPrice - 20;
}
```

Do the two statements always compute the same value? If not, when do the values differ?

4. Consider the following statements to compute a discounted price:

```
discountedPrice = originalPrice;
if (originalPrice > 100)
{
    discountedPrice = originalPrice - 10;
}
```

What is the discounted price if the original price is 95? 100? 105?

5. The variables fuelAmount and fuelCapacity hold the actual amount of fuel and the size of the fuel tank of a vehicle. If less than 10 percent is remaining in the tank, a status light should show a red color; otherwise it shows a green color. Simulate this process by printing out either "red" or "green".

Practice It Now you can try these exercises at the end of the chapter: R3.5, R3.6, P3.17.

Programming Tip 3.1**Brace Layout**

The compiler doesn't care where you place braces. In this book, we follow the simple rule of making { and } line up.

```
if (floor > 13)
{
    floor--;
}
```

This style makes it easy to spot matching braces. Some programmers put the opening brace on the same line as the if:

```
if (floor > 13) {
    floor--;
}
```



© Timothy Large/Stockphoto.

Properly lining up your code makes your programs easier to read.

This style makes it harder to match the braces, but it saves a line of code, allowing you to view more code on the screen without scrolling. There are passionate advocates of both styles.

It is important that you pick a layout style and stick with it consistently within a given programming project. Which style you choose may depend on your personal preference or a coding style guide that you need to follow.

Programming Tip 3.2**Always Use Braces**

When the body of an if statement consists of a single statement, you need not use braces. For example, the following is legal:

```
if (floor > 13)
    floor--;
```

However, it is a good idea to always include the braces:

```
if (floor > 13)
{
    floor--;
}
```

The braces make your code easier to read. They also make it easier for you to maintain the code because you won't have to worry about adding braces when you add statements inside an if statement.

Common Error 3.1**A Semicolon After the if Condition**

The following code fragment has an unfortunate error:

```
if (floor > 13) ; // ERROR
{
    floor--;
}
```

There should be no semicolon after the if condition. The compiler interprets this statement as follows: If floor is greater than 13, execute the statement that is denoted by a single semicolon, that is, the do-nothing statement. The statement enclosed in braces is no longer a part of the if

statement. It is always executed. In other words, even if the value of `floor` is not above 13, it is decremented.

Programming Tip 3.3



Tabs

Block-structured code has the property that nested statements are indented by one or more levels:

```
public class ElevatorSimulation
{
    public static void main(String[] args)
    {
        int floor;
        . .
        if (floor > 13)
        {
            floor--;
        }
        . .
    }
    | | |
0 1 2 3   Indentation level
```



You use
the Tab key
to move the
cursor to the next
indentation level.

Photo by Vincent LaRussa/John Wiley & Sons, Inc.

How do you move the cursor from the leftmost column to the appropriate indentation level? A perfectly reasonable strategy is to hit the space bar a sufficient number of times. With most editors, you can use the Tab key instead. A tab moves the cursor to the next indentation level. Some editors even have an option to fill in the tabs automatically.

While the Tab key is nice, some editors use *tab characters* for alignment, which is not so nice. Tab characters can lead to problems when you send your file to another person or a printer. There is no universal agreement on the width of a tab character, and some software will ignore tab characters altogether. It is therefore best to save your files with spaces instead of tabs. Most editors have a setting to automatically convert all tabs to spaces. Look at the documentation of your development environment to find out how to activate this useful setting.

Special Topic 3.1



The Conditional Operator

Java has a *conditional operator* of the form

`condition ? value1 : value2`

The value of that expression is either `value1` if the test passes or `value2` if it fails. For example, we can compute the actual floor number as

`actualFloor = floor > 13 ? floor - 1 : floor;`

which is equivalent to

`if (floor > 13) { actualFloor = floor - 1; } else { actualFloor = floor; }`

You can use the conditional operator anywhere that a value is expected, for example:

`System.out.println("Actual floor: " + (floor > 13 ? floor - 1 : floor));`

We don't use the conditional operator in this book, but it is a convenient construct that you will find in many Java programs.

Programming Tip 3.4

**Avoid Duplication in Branches**

Look to see whether you *duplicate code* in each branch. If so, move it out of the if statement. Here is an example of such duplication:

```
if (floor > 13)
{
    actualFloor = floor - 1;
    System.out.println("Actual floor: " + actualFloor);
}
else
{
    actualFloor = floor;
    System.out.println("Actual floor: " + actualFloor);
}
```

The output statement is exactly the same in both branches. This is not an error—the program will run correctly. However, you can simplify the program by moving the duplicated statement, like this:

```
if (floor > 13)
{
    actualFloor = floor - 1;
}
else
{
    actualFloor = floor;
}
System.out.println("Actual floor: " + actualFloor);
```

Removing duplication is particularly important when programs are maintained for a long time. When there are two sets of statements with the same effect, it can easily happen that a programmer modifies one set but not the other.

3.2 Comparing Numbers and Strings

Use relational operators ($<$ \leq $>$ \geq $==$ $!=$) to compare numbers.

Every if statement contains a condition. In many cases, the condition involves comparing two values. For example, in the previous examples we tested `floor > 13`. The comparison `>` is called a **relational operator**. Java has six relational operators (see Table 1).

As you can see, only two Java relational operators (`>` and `<`) look as you would expect from the mathematical notation. Computer keyboards do not have keys for \geq , \leq , or \neq , but the `\geq` , `\leq` , and `\neq` operators are easy to remember because they look similar. The `$==$` operator is initially confusing to most newcomers to Java.



© arturboi/Stockphoto.

In Java, you use a relational operator to check whether one value is greater than another.

Table 1 Relational Operators

Java	Math Notation	Description
>	>	Greater than
\geq	\geq	Greater than or equal
<	<	Less than
\leq	\leq	Less than or equal
\equiv	=	Equal
\neq	\neq	Not equal

In Java, = already has a meaning, namely assignment. The == operator denotes equality testing:

```
floor = 13; // Assign 13 to floor
if (floor == 13) // Test whether floor equals 13
```

You must remember to use == inside tests and to use = outside tests.

Syntax 3.2 Comparisons

Check that you have the right direction:
 > (greater) or < (less)

These quantities are compared.

floor > 13

One of: == != < <= > >= (See Table 1.)

Check the boundary condition:
 > (greater) or \geq (greater or equal)?

floor == 13

Checks for equality.

Use ==, not =.

```
String input;
if (input.equals("Y"))
```

Use equals to compare strings. (See Common Error 3.3.)

```
double x; double y; final double EPSILON = 1E-14;
if (Math.abs(x - y) < EPSILON)
```

Checks that these floating-point numbers are very close.
 See Common Error 3.2.

**FULL CODE EXAMPLE**

Go to wiley.com/go/bjlo2code to download a program that demonstrates comparisons of numbers and strings.

Do not use the `==` operator to compare strings. Use the `equals` method instead.

The relational operators in Table 1 have a lower precedence than the arithmetic operators. That means, you can write arithmetic expressions on either side of the relational operator without using parentheses. For example, in the expression

`floor - 1 < 13`

both sides (`floor - 1` and `13`) of the `<` operator are evaluated, and the results are compared. Appendix B shows a table of the Java operators and their precedence.

To test whether two strings are equal to each other, you must use the method called `equals`:

```
if (string1.equals(string2)) . . .
```

Do not use the `==` operator to compare strings. The comparison

```
if (string1 == string2) // Not useful
```

has an unrelated meaning. It tests whether the two strings are stored in the same location. You can have strings with identical contents stored in different locations, so this test never makes sense in actual programming; see Common Error 3.3 on page 94.

Table 2 summarizes how to compare values in Java.

Table 2 Relational Operator Examples

Expression	Value	Comment
<code>3 <= 4</code>	true	3 is less than 4; <code><=</code> tests for “less than or equal”.
<code>3 = < 4</code>	Error	The “less than or equal” operator is <code><=</code> , not <code>=<</code> . The “less than” symbol comes first.
<code>3 > 4</code>	false	<code>></code> is the opposite of <code><=</code> .
<code>4 < 4</code>	false	The left-hand side must be strictly smaller than the right-hand side.
<code>4 <= 4</code>	true	Both sides are equal; <code><=</code> tests for “less than or equal”.
<code>3 == 5 - 2</code>	true	<code>==</code> tests for equality.
<code>3 != 5 - 1</code>	true	<code>!=</code> tests for inequality. It is true that 3 is not $5 - 1$.
<code>3 = 6 / 2</code>	Error	Use <code>==</code> to test for equality.
<code>1.0 / 3.0 == 0.333333333</code>	false	Although the values are very close to one another, they are not exactly equal. See Common Error 3.2 on page 93.
<code>"10" > 5</code>	Error	You cannot compare a string to a number.
<code>"Tomato".substring(0, 3).equals("Tom")</code>	true	Always use the <code>equals</code> method to check whether two strings have the same contents.
<code>"Tomato".substring(0, 3) == ("Tom")</code>	false	Never use <code>==</code> to compare strings; it only checks whether the strings are stored in the same location. See Common Error 3.3 on page 94.

SELF CHECK

6. Which of the following conditions are true, provided a is 3 and b is 4?
 - $a + 1 \leq b$
 - $a + 1 \geq b$
 - $a + 1 \neq b$
7. Give the opposite of the condition
 $\text{floor} > 13$
8. What is the error in this statement?

```
if (scoreA = scoreB)
{
    System.out.println("Tie");
}
```
9. Supply a condition in this if statement to test whether the user entered a Y:

```
System.out.println("Enter Y to quit.");
String input = in.next();
if (. . .)
{
    System.out.println("Goodbye.");
}
```
10. How do you test that a string str is the empty string?

Practice It Now you can try these exercises at the end of the chapter: R3.4, R3.7, E3.14.

Common Error 3.2**Exact Comparison of Floating-Point Numbers**

Floating-point numbers have only a limited precision, and calculations can introduce roundoff errors. You must take these inevitable roundoffs into account when comparing floating-point numbers. For example, the following code multiplies the square root of 2 by itself. Ideally, we expect to get the answer 2:

```
double r = Math.sqrt(2.0);
if (r * r == 2.0)
{
    System.out.println("Math.sqrt(2.0) squared is 2.0");
}
else
{
    System.out.println("Math.sqrt(2.0) squared is not 2.0 but "
        + r * r);
}
```

This program displays

`Math.sqrt(2.0) squared is not 2.0 but 2.0000000000000044`

It does not make sense in most circumstances to compare floating-point numbers exactly. Instead, we should test whether they are *close enough*. That is, the magnitude of their difference should be less than some threshold. Mathematically, we would write that x and y are close enough if

$$|x - y| < \varepsilon$$



Take limited precision into account when comparing floating-point numbers.

for a very small number, ε . ε is the Greek letter epsilon, a letter used to denote a very small quantity. It is common to set ε to 10^{-14} when comparing double numbers:

```
final double EPSILON = 1E-14;
double r = Math.sqrt(2.0);
if (Math.abs(r * r - 2.0) < EPSILON)
{
    System.out.println("Math.sqrt(2.0) squared is approximately 2.0");
}
```

Common Error 3.3



Using == to Compare Strings

If you write

```
if (nickname == "Rob")
```

then the test succeeds only if the variable nickname refers to the exact same location as the string literal "Rob". The test will pass if a string variable was initialized with the same string literal:

```
String nickname = "Rob";
if (nickname == "Rob") // Test is true
```

However, if the string with the letters R o b has been assembled in some other way, then the test will fail:

```
String name = "Robert";
String nickname = name.substring(0, 3);
...
if (nickname == "Rob") // Test is false
```

In this case, the substring method produces a string in a different memory location. Even though both strings have the same contents, the comparison fails.

You must remember never to use == to compare strings. Always use equals to check whether two strings have the same contents.

Special Topic 3.2



Lexicographic Ordering of Strings

If two strings are not identical to each other, you still may want to know the relationship between them. The compareTo method compares strings in “lexicographic” order. This ordering is very similar to the way in which words are sorted in a dictionary. If

```
string1.compareTo(string2) < 0
```

then the string string1 comes before the string string2 in the dictionary. For example, this is the case if string1 is "Harry", and string2 is "Hello". If

```
string1.compareTo(string2) > 0
```

then string1 comes after string2 in dictionary order.

Finally, if

```
string1.compareTo(string2) == 0
```

then string1 and string2 are equal.



Corbis Digital Stock.

To see which of two terms comes first in the dictionary, consider the first letter in which they differ.

There are a few technical differences between the ordering in a dictionary and the lexicographic ordering in Java. In Java:

- All uppercase letters come before the lowercase letters. For example, "z" comes before "a".
- The space character comes before all printable characters.
- Numbers come before letters.
- For the ordering of punctuation marks, see Appendix A.

The `compareTo` method compares strings in lexicographic order.

When comparing two strings, you compare the first letters of each word, then the second letters, and so on, until one of the strings ends or you find the first letter pair that doesn't match.

If one of the strings ends, the longer string is considered the “larger” one. For example, compare “car” with “cart”. The first three letters match, and we reach the end of the first string. Therefore “car” comes before “cart” in lexicographic ordering.

When you reach a mismatch, the string containing the “larger” character is considered “larger”. For example, let's compare “cat” with “cart”. The first two letters match. Because t comes after r, the string “cat” comes after “cart” in the lexicographic ordering.

c a r

c a r t

c a t

Letters r comes
match before t

Lexicographic
Ordering

HOW TO 3.1

Implementing an if Statement



This How To walks you through the process of implementing an `if` statement. We will illustrate the steps with the following example problem.

Problem Statement The university bookstore has a Kilobyte Day sale every October 24, giving an 8 percent discount on all computer accessory purchases if the price is less than \$128, and a 16 percent discount if the price is at least \$128. Write a program that asks the cashier for the original price and then prints the discounted price.

Step 1 Decide upon the branching condition.

In our sample problem, the obvious choice for the condition is:

`original price < 128?`

That is just fine, and we will use that condition in our solution.

But you could equally well come up with a correct solution if you choose the opposite condition: Is the original price at least \$128? You might choose this condition if you put yourself into the position of a shopper who wants to know when the bigger discount applies.



© MikePanic/Stockphoto.

Sales discounts are often higher for expensive products. Use the `if` statement to implement such a decision.

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

In this step, you list the action or actions that are taken in the “positive” branch. The details depend on your problem. You may want to print a message, compute values, or even exit the program.

In our example, we need to apply an 8 percent discount:

$$\text{discounted price} = 0.92 \times \text{original price}$$

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

What do you want to do in the case that the condition of Step 1 is not satisfied? Sometimes, you want to do nothing at all. In that case, use an `if` statement without an `else` branch.

In our example, the condition tested whether the price was less than \$128. If that condition is *not* true, the price is at least \$128, so the higher discount of 16 percent applies to the sale:

$$\text{discounted price} = 0.84 \times \text{original price}$$

Step 4 Double-check relational operators.

First, be sure that the test goes in the right *direction*. It is a common error to confuse `>` and `<`. Next, consider whether you should use the `<` operator or its close cousin, the `<=` operator.

What should happen if the original price is exactly \$128? Reading the problem carefully, we find that the lower discount applies if the original price is *less than* \$128, and the higher discount applies when it is *at least* \$128. A price of \$128 should therefore *not* fulfill our condition, and we must use `<`, not `<=`.

Step 5 Remove duplication.

Check which actions are common to both branches, and move them outside. (See Programming Tip 3.4 on page 90.)

In our example, we have two statements of the form

$$\text{discounted price} = \underline{\quad} \times \text{original price}$$

They only differ in the discount rate. It is best to just set the rate in the branches, and to do the computation afterwards:

```
If original price < 128
    discount rate = 0.92
Else
    discount rate = 0.84
discounted price = discount rate × original price
```

Step 6 Test both branches.

Formulate two test cases, one that fulfills the condition of the `if` statement, and one that does not. Ask yourself what should happen in each case. Then follow the pseudocode and act each of them out.

In our example, let us consider two scenarios for the original price: \$100 and \$200. We expect that the first price is discounted by \$8, the second by \$32.

When the original price is 100, then the condition `100 < 128` is true, and we get

```
discount rate = 0.92
discounted price = 0.92 × 100 = 92
```

When the original price is 200, then the condition `200 < 128` is false, and

```
discount rate = 0.84
discounted price = 0.84 × 200 = 168
```

In both cases, we get the expected answer.

Step 7 Assemble the `if` statement in Java.

Type the skeleton

```
if ()
{
}
else
{
```

}

and fill it in, as shown in Syntax 3.1 on page 86. Omit the `else` branch if it is not needed.

In our example, the completed statement is

```
if (originalPrice < 128)
{
    discountRate = 0.92;
}
else
{
    discountRate = 0.84;
}
discountedPrice = discountRate * originalPrice;
```



FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download the program for calculating a discounted price.



WORKED EXAMPLE 3.1



Extracting the Middle

Learn how to extract the middle character from a string, or the two middle characters if the length of the string is even. Go to wiley.com/go/bjlo2examples and download Worked Example 3.1.

c	r	a	t	e
0	1	2	3	4



Computing & Society 3.1 The Denver Airport Luggage System

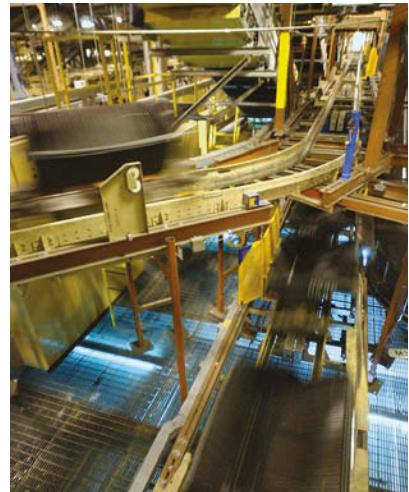
Making decisions is an essential part of any computer program. Nowhere is this more obvious than in a computer system that helps sort luggage at an airport. After scanning the luggage identification codes, the system sorts the items and routes them to different conveyor belts. Human operators then place the items onto trucks. When the city of Denver built a huge airport to replace an outdated and congested facility, the luggage system contractor went a step further. The new system was designed to replace the human operators with robotic carts. Unfortunately, the system plainly did not work. It was plagued by mechanical problems, such as luggage falling onto the tracks and jamming carts. Equally frustrating were the software glitches. Carts would uselessly accumulate at some locations when they were needed elsewhere.

The airport had been scheduled to open in 1993, but without a functioning luggage system, the opening was delayed for over a year while the

contractor tried to fix the problems. The contractor never succeeded, and ultimately a manual system was installed. The delay cost the city and airlines close to a billion dollars, and the contractor, once the leading luggage systems vendor in the United States, went bankrupt.

Clearly, it is very risky to build a large system based on a technology that has never been tried on a smaller scale. In 2013, the rollout of universal healthcare in the United States was put in jeopardy by a dysfunctional web site for selecting insurance plans. The system promised an insurance shopping experience similar to booking airline flights. But, the HealthCare.gov site didn't simply present the available insurance plans. It also had to check the income level of each applicant and use that information to determine the subsidy level. That task turned out to be quite a bit harder than checking whether a credit card had sufficient credit to pay for an airline ticket. The Obama administration would have been well advised to design a signup

process that did not rely on an untested computer program.



Lyn Alweis/The Denver Post via Getty Images Inc.

The Denver airport originally had a fully automatic system for moving luggage, replacing human operators with robotic carts. Unfortunately, the system never worked and was dismantled before the airport was opened.

3.3 Multiple Alternatives

Multiple if statements can be combined to evaluate complex decisions.

The 1989 Loma Prieta earthquake that damaged the Bay Bridge in San Francisco and destroyed many buildings measured 7.1 on the Richter scale.



© kevinussi/Stockphoto.

In Section 3.1, you saw how to program a two-way branch with an `if` statement. In many situations, there are more than two cases. In this section, you will see how to implement a decision with multiple alternatives.

For example, consider a program that displays the effect of an earthquake, as measured by the Richter scale (see Table 3).

Table 3 Richter Scale

Value	Effect
8	Most structures fall
7	Many buildings destroyed
6	Many buildings considerably damaged, some collapse
4.5	Damage to poorly constructed buildings

The Richter scale is a measurement of the strength of an earthquake. Every step in the scale, for example from 6.0 to 7.0, signifies a tenfold increase in the strength of the quake.

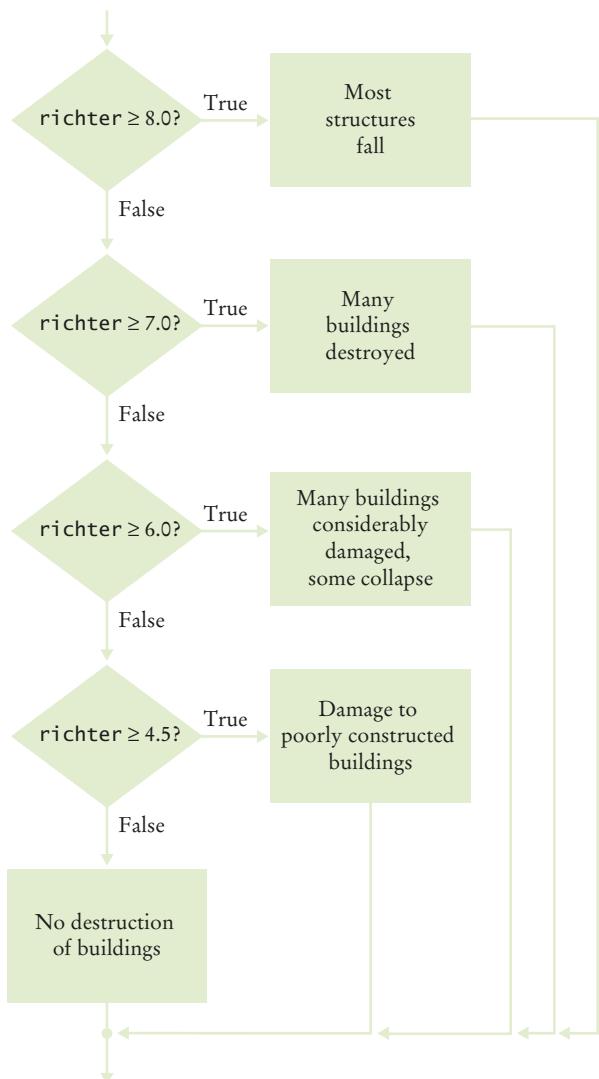
In this case, there are five branches: one each for the four descriptions of damage, and one for no destruction. Figure 3 shows the flowchart for this multiple-branch statement.

You use multiple `if` statements to implement multiple alternatives, like this:

```
if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some collapse");
}
else if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}
else
{
    System.out.println("No destruction of buildings");
}
```

As soon as one of the four tests succeeds, the effect is displayed, and no further tests are attempted. If none of the four cases applies, the final `else` clause applies, and a default message is printed.

Figure 3
Multiple Alternatives



Here you must sort the conditions and test against the largest cutoff first.
Suppose we reverse the order of tests:

```

if (richter >= 4.5) // Tests in wrong order
{
    System.out.println("Damage to poorly constructed buildings");
}
else if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some collapse");
}
else if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
  
```

```

    }
else if (richter >= 8.0)
{
    System.out.println("Most structures fall");
}

```

When using multiple if statements, test general conditions after more specific conditions.

This does not work. Suppose the value of richter is 7.1. That value is at least 4.5, matching the first case. The other tests will never be attempted.

The remedy is to test the more specific conditions first. Here, the condition richter ≥ 8.0 is more specific than the condition richter ≥ 7.0 , and the condition richter ≥ 4.5 is more general (that is, fulfilled by more values) than either of the first two.

In this example, it is also important that we use an if/else if/else sequence, not just multiple independent if statements. Consider this sequence of independent tests.

```

if (richter >= 8.0) // Didn't use else
{
    System.out.println("Most structures fall");
}
if (richter >= 7.0)
{
    System.out.println("Many buildings destroyed");
}
if (richter >= 6.0)
{
    System.out.println("Many buildings considerably damaged, some collapse");
}
if (richter >= 4.5)
{
    System.out.println("Damage to poorly constructed buildings");
}

```

Now the alternatives are no longer exclusive. If richter is 7.1, then the last *three* tests all match, and three messages are printed.

FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download the program for printing earthquake descriptions.



SELF CHECK

11. In a game program, the scores of players A and B are stored in variables scoreA and scoreB. Assuming that the player with the larger score wins, write an if/else if/else sequence that prints out "A won", "B won", or "Game tied".
12. Write a conditional statement with three branches that sets s to 1 if x is positive, to -1 if x is negative, and to 0 if x is zero.
13. How could you achieve the task of Self Check 12 with only two branches?
14. Beginners sometimes write statements such as the following:

```

if (price > 100)
{
    discountedPrice = price - 20;
}
else if (price <= 100)
{
    discountedPrice = price - 10;
}

```

Explain how this code can be improved.

15. Suppose the user enters -1 into the earthquake program. What is printed?

- 16.** Suppose we want to have the earthquake program check whether the user entered a negative number. What branch would you add to the `if` statement, and where?

Practice It Now you can try these exercises at the end of the chapter: R3.23, E3.10, P3.20.

Special Topic 3.3



The switch Statement

An `if/else if/else` sequence that compares a *value* against several alternatives can be implemented as a `switch` statement. For example,

```
int digit = . . .;
switch (digit)
{
    case 1: digitName = "one"; break;
    case 2: digitName = "two"; break;
    case 3: digitName = "three"; break;
    case 4: digitName = "four"; break;
    case 5: digitName = "five"; break;
    case 6: digitName = "six"; break;
    case 7: digitName = "seven"; break;
    case 8: digitName = "eight"; break;
    case 9: digitName = "nine"; break;
    default: digitName = ""; break;
}
```

This is a shortcut for

```
int digit = . . .;
if (digit == 1) { digitName = "one"; }
else if (digit == 2) { digitName = "two"; }
else if (digit == 3) { digitName = "three"; }
else if (digit == 4) { digitName = "four"; }
else if (digit == 5) { digitName = "five"; }
else if (digit == 6) { digitName = "six"; }
else if (digit == 7) { digitName = "seven"; }
else if (digit == 8) { digitName = "eight"; }
else if (digit == 9) { digitName = "nine"; }
else { digitName = ""; }
```

It isn't much of a shortcut, but it has one advantage—it is obvious that all branches test the *same* value, namely `digit`.

The `switch` statement can be applied only in narrow circumstances. The values in the `case` clauses must be constants. They can be integers or characters. As of Java 7, strings are permitted as well. You cannot use a `switch` statement to branch on floating-point values.

Every branch of the `switch` should be terminated by a `break` instruction. If the `break` is missing, execution *falls through* to the next branch, and so on, until a `break` or the end of the `switch` is reached. In practice, this fall-through behavior is rarely useful, but it is a common cause of errors. If you accidentally forget a `break` statement, your program compiles but executes unwanted code. Many programmers consider the `switch` statement somewhat dangerous and prefer the `if` statement.

We leave it to you to use the `switch` statement for your own code or not. At any rate, you need to have a reading knowledge of `switch` in case you find it in other programmers' code.



© travelpixpro/iStockphoto.

The switch statement lets you choose from a fixed set of alternatives.

3.4 Nested Branches

When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.

It is often necessary to include an `if` statement inside another. Such an arrangement is called a *nested* set of statements.

Here is a typical example: In the United States, different tax rates are used depending on the taxpayer's marital status. There are different tax schedules for single and for married taxpayers. Married taxpayers add their income together and pay taxes on the total. Table 4 gives the tax rate computations, using a simplification of the schedules in effect for the 2008 tax year. A different tax rate applies to each "bracket". In this schedule, the income in the first bracket is taxed at 10 percent, and the income in the second bracket is taxed at 25 percent. The income limits for each bracket depend on the marital status.

Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most \$32,000	10%	\$0
over \$32,000	$\$3,200 + 25\%$	\$32,000
If your status is Married and if the taxable income is	the tax is	of the amount over
at most \$64,000	10%	\$0
over \$64,000	$\$6,400 + 25\%$	\$64,000

Nested decisions are required for problems that have two levels of decision making.

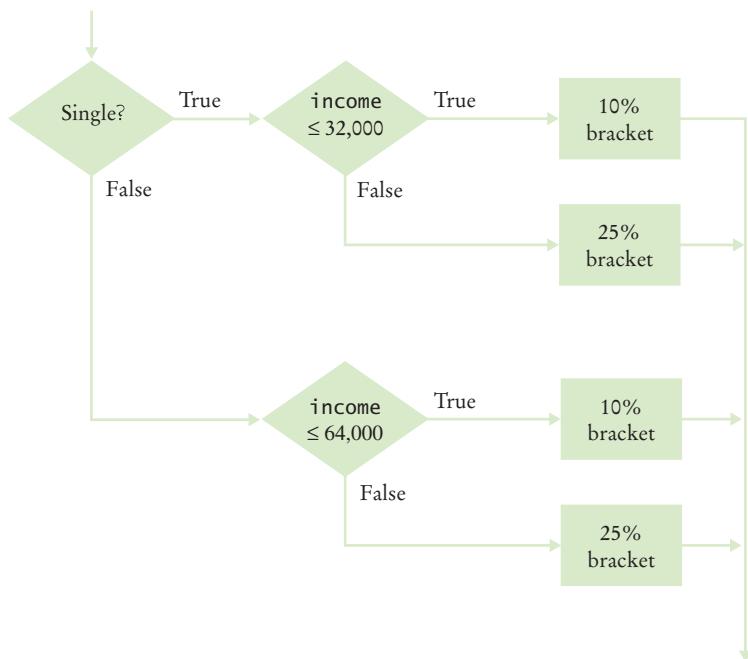
Now compute the taxes due, given a marital status and an income figure. The key point is that there are two *levels* of decision making. First, you must branch on the marital status. Then, for each marital status, you must have another branch on income level.

The two-level decision process is reflected in two levels of `if` statements in the program at the end of this section. (See Figure 4 for a flowchart.) In theory, nesting can go deeper than two levels. A three-level decision process (first by state, then by marital status, then by income level) requires three nesting levels.



Computing income taxes requires multiple levels of decisions.

© ericsphotography/iStockphoto.

**Figure 4** Income Tax Computation**sec04/TaxCalculator.java**

```

1 import java.util.Scanner;
2
3 /**
4  * This program computes income taxes, using a simplified tax schedule.
5 */
6 public class TaxCalculator
7 {
8     public static void main(String[] args)
9     {
10         final double RATE1 = 0.10;
11         final double RATE2 = 0.25;
12         final double RATE1_SINGLE_LIMIT = 32000;
13         final double RATE1_MARRIED_LIMIT = 64000;
14
15         double tax1 = 0;
16         double tax2 = 0;
17
18         // Read income and marital status
19
20         Scanner in = new Scanner(System.in);
21         System.out.print("Please enter your income: ");
22         double income = in.nextDouble();
23
24         System.out.print("Please enter s for single, m for married: ");
25         String maritalStatus = in.next();
26
27         // Compute taxes due
28
  
```

```

29     if (maritalStatus.equals("s"))
30     {
31         if (income <= RATE1_SINGLE_LIMIT)
32         {
33             tax1 = RATE1 * income;
34         }
35     else
36     {
37         tax1 = RATE1 * RATE1_SINGLE_LIMIT;
38         tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
39     }
40 }
41 else
42 {
43     if (income <= RATE1_MARRIED_LIMIT)
44     {
45         tax1 = RATE1 * income;
46     }
47 else
48 {
49     tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50     tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51 }
52 }
53
54 double totalTax = tax1 + tax2;
55
56 System.out.println("The tax is $" + totalTax);
57 }
58 }
```

Program Run

```

Please enter your income: 80000
Please enter s for single, m for married: m
The tax is $10400
```



SELF CHECK

17. What is the amount of tax that a single taxpayer pays on an income of \$32,000?
18. Would that amount change if the first nested if statement changed from
`if (income <= RATE1_SINGLE_LIMIT)`
to
`if (income < RATE1_SINGLE_LIMIT)`
19. Suppose Harry and Sally each make \$40,000 per year. Would they save taxes if they married?
20. How would you modify the `TaxCalculator.java` program in order to check that the user entered a correct value for the marital status (i.e., `s` or `m`)?
21. Some people object to higher tax rates for higher incomes, claiming that you might end up with less money after taxes when you get a raise for working hard. What is the flaw in this argument?

Practice It Now you can try these exercises at the end of the chapter: R3.10, R3.22, E3.14, P3.7.

Programming Tip 3.5

**Hand-Tracing**

A very useful technique for understanding whether a program works correctly is called *hand-tracing*. You simulate the program's activity on a sheet of paper. You can use this method with pseudocode or Java code.

Get an index card, a cocktail napkin, or whatever sheet of paper is within reach. Make a column for each variable. Have the program code ready. Use a marker, such as a paper clip, to mark the current statement. In your mind, execute statements one at a time. Every time the value of a variable changes, cross out the old value and write the new value below the old one.

For example, let's trace the tax program with the data from the program run on page 104. In lines 15 and 16, tax1 and tax2 are initialized to 0.

```

8 public static void main(String[] args)
9 {
10    final double RATE1 = 0.10;
11    final double RATE2 = 0.25;
12    final double RATE1_SINGLE_LIMIT = 32000;
13    final double RATE1_MARRIED_LIMIT = 64000;
14
15    double tax1 = 0;
16    double tax2 = 0;
17

```

In lines 22 and 25, income and maritalStatus are initialized by input statements.

```

20   Scanner in = new Scanner(System.in);
21   System.out.print("Please enter your income: ");
22   double income = in.nextDouble();
23
24   System.out.print("Please enter s for single, m for married: ");
25   String maritalStatus = in.next();

```

Because maritalStatus is not "s", we move to the else branch of the outer if statement (line 41).

```

29   if (maritalStatus.equals("s"))
30   {
31     if (income <= RATE1_SINGLE_LIMIT)
32     {
33       tax1 = RATE1 * income;
34     }
35     else
36     {
37       tax1 = RATE1 * RATE1_SINGLE_LIMIT;
38       tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT);
39     }
40   }
41   else
42   {

```

Because income is not <= 64000, we move to the else branch of the inner if statement (line 47).

```

43   if (income <= RATE1_MARRIED_LIMIT)
44   {
45     tax1 = RATE1 * income;
46   }
47   else
48   {
49     tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50     tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51   }

```



© thomasd007/iStockphoto.

Hand-tracing helps you understand whether a program works correctly.

tax1	tax2	income	marital status
0	0		

tax1	tax2	income	marital status
0	0	80000	m

The values of tax1 and tax2 are updated.

```

48      {
49          tax1 = RATE1 * RATE1_MARRIED_LIMIT;
50          tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT);
51      }
52  }
53

```

tax1	tax2	income	marital status
0	0	80000	m
6400	4000		

Their sum totalTax is computed and printed. Then the program ends.

```

54     double totalTax = tax1 + tax2;
55
56     System.out.println("The tax is $" + totalTax);
57 }

```

Because the program trace shows the expected output (\$10,400), it successfully demonstrated that this test case works correctly.

tax1	tax2	income	marital status	total tax
0	0	80000	m	
6400	4000			10400

Common Error 3.4



The Dangling else Problem

When an if statement is nested inside another if statement, the following error may occur.

```

double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is more expensive
else // Pitfall!
    shippingCharge = 20.00; // As are foreign shipments

```

The indentation level seems to suggest that the else is grouped with the test country.equals("USA"). Unfortunately, that is not the case. The compiler ignores all indentation and matches the else with the preceding if. That is, the code is actually

```

double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
    if (state.equals("HI"))
        shippingCharge = 10.00; // Hawaii is more expensive
    else // Pitfall!
        shippingCharge = 20.00; // As are foreign shipments

```

That isn't what you want. You want to group the else with the first if.

The ambiguous else is called a *dangling else*. You can avoid this pitfall if you always use braces, as recommended in Programming Tip 3.2 on page 88:

```

double shippingCharge = 5.00; // $5 inside continental U.S.
if (country.equals("USA"))
{
    if (state.equals("HI"))
    {
        shippingCharge = 10.00; // Hawaii is more expensive
    }
}
else
{
    shippingCharge = 20.00; // As are foreign shipments
}

```

Special Topic 3.4

**Enumeration Types**

In many programs, you use variables that can hold one of a finite number of values. For example, in the tax return class, the `maritalStatus` variable holds one of the values "s" or "m". If, due to some programming error, the `maritalStatus` variable is set to another value (such as "d" or "w"), then the programming logic may produce invalid results.

In a simple program, this is not really a problem. But as programs grow over time, and more cases are added (such as the “married filing separately” status), errors can slip in. **Enumeration types** provide a remedy. An enumeration type has a finite set of values, for example

```
public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }
```

You can have any number of values, but you must include them all in the `enum` declaration.

You can declare variables of the enumeration type:

```
FilingStatus status = FilingStatus.SINGLE;
```

If you try to assign a value that isn’t a `FilingStatus`, such as 2 or "S", then the compiler reports an error.

Use the `==` operator to compare enumeration values, for example:

```
if (status == FilingStatus.SINGLE) . . .
```

Place the `enum` declaration inside the class that implements your program, such as

```
public class TaxReturn
{
    public enum FilingStatus { SINGLE, MARRIED, MARRIED_FILING_SEPARATELY }

    public static void main(String[] args)
    {
        . . .
    }
}
```



VIDEO EXAMPLE 3.1

Computing the Plural of an English Word

The plural of apple is apples, but the plural of cherry is cherries. In this Video Example, we develop an algorithm for computing the plural of an English word. Go to wiley.com/go/bjlo2videos to view Video Example 3.1.



© mikie11/iStockphoto.

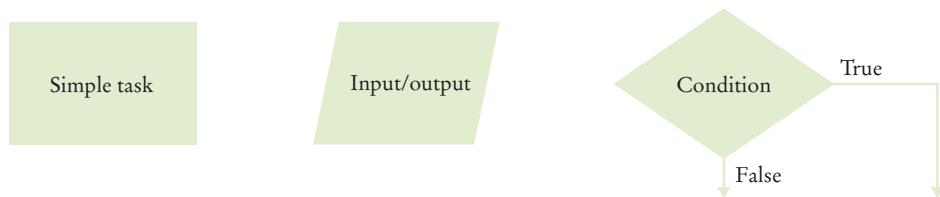
3.5 Problem Solving: Flowcharts

Flowcharts are made up of elements for tasks, input/output, and decisions.

You have seen examples of flowcharts earlier in this chapter. A flowchart shows the structure of decisions and tasks that are required to solve a problem. When you have to solve a complex problem, it is a good idea to draw a flowchart to visualize the flow of control.

The basic flowchart elements are shown in Figure 5.

Figure 5
Flowchart Elements



Each branch of a decision can contain tasks and further decisions.

The basic idea is simple enough. Link tasks and input/output boxes in the sequence in which they should be executed. Whenever you need to make a decision, draw a diamond with two outcomes (see Figure 6).

Each branch can contain a sequence of tasks and even additional decisions. If there are multiple choices for a value, lay them out as in Figure 7.

There is one issue that you need to be aware of when drawing flowcharts. Unconstrained branching and merging can lead to “spaghetti code”, a messy network of possible pathways through a program.

There is a simple rule for avoiding spaghetti code: Never point an arrow *inside another branch*.

To understand the rule, consider this example: Shipping costs are \$5 inside the United States, except that to Hawaii and Alaska they are \$10. International shipping costs are also \$10.

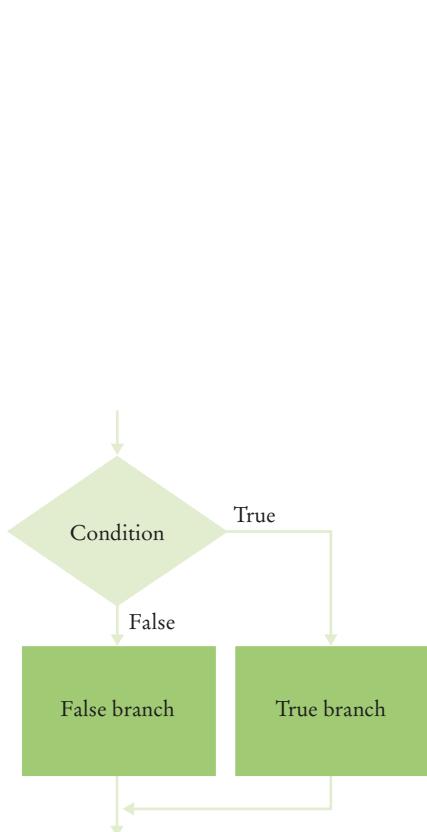


Figure 6 Flowchart with Two Outcomes

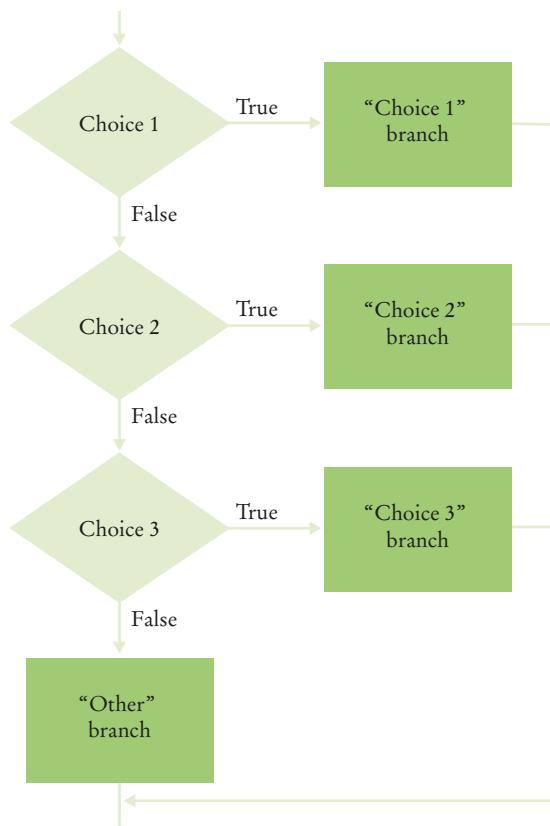
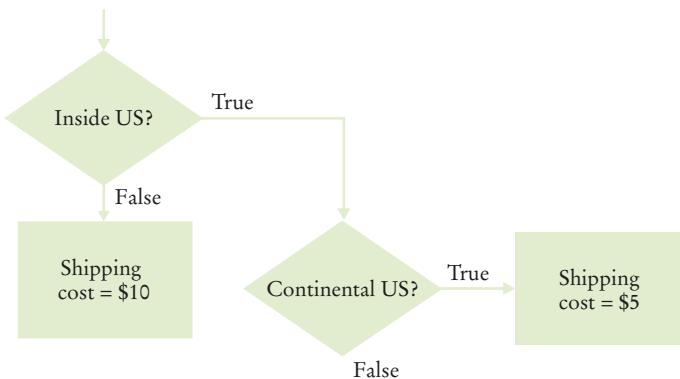
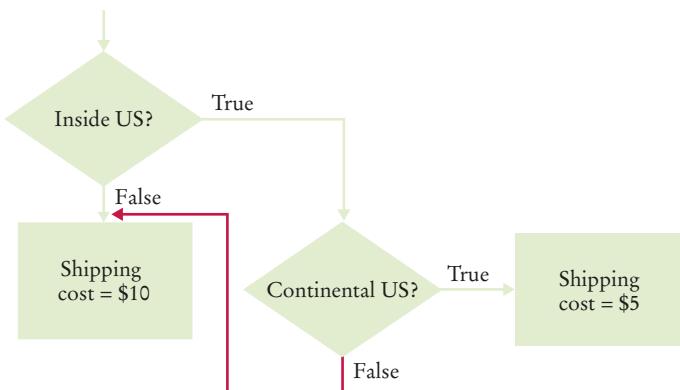


Figure 7 Flowchart with Multiple Choices

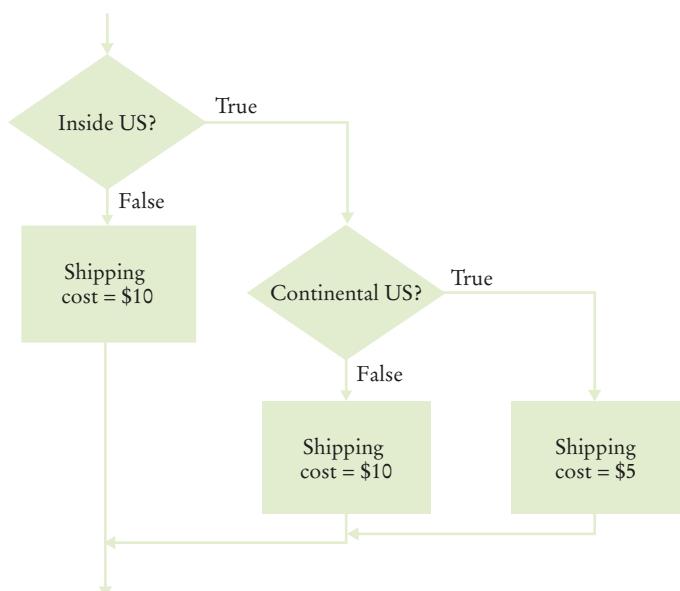
You might start out with a flowchart like the following:



Now you may be tempted to reuse the “shipping cost = \$10” task:



Don’t do that! The red arrow points inside a different branch. Instead, add another task that sets the shipping cost to \$10, like this:



**FULL CODE EXAMPLE**

Go to wiley.com/go/bjlo2code to download a program to compute shipping costs.

Not only do you avoid spaghetti code, but it is also a better design. In the future it may well happen that the cost for international shipments is different from that to Alaska and Hawaii.

Flowcharts can be very useful for getting an intuitive understanding of the flow of an algorithm. However, they get large rather quickly when you add more details. At that point, it makes sense to switch from flowcharts to pseudocode.

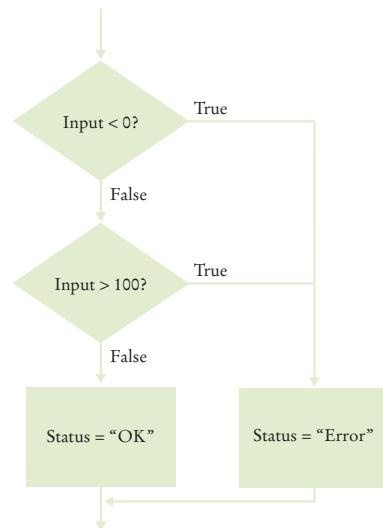


© Ekspansio/iStockphoto.

Spaghetti code has so many pathways that it becomes impossible to understand.

**SELF CHECK**

22. Draw a flowchart for a program that reads a value `temp` and prints “Frozen” if it is less than zero.
23. What is wrong with the flowchart at right?
24. How do you fix the flowchart of Self Check 23?
25. Draw a flowchart for a program that reads a value `x`. If it is less than zero, print “Error”. Otherwise, print its square root.
26. Draw a flowchart for a program that reads a value `temp`. If it is less than zero, print “Ice”. If it is greater than 100, print “Steam”. Otherwise, print “Liquid”.

**Practice It**

Now you can try these exercises at the end of the chapter: R3.13, R3.14, R3.15.

3.6 Problem Solving: Test Cases

Consider how to test the tax computation program from Section 3.4. Of course, you cannot try out all possible inputs of marital status and income level. Even if you could, there would be no point in trying them all. If the program correctly computes one or two tax amounts in a given bracket, then we have a good reason to believe that all amounts will be correct.

You want to aim for complete *coverage* of all decision points. Here is a plan for obtaining a comprehensive set of test cases:

- There are two possibilities for the marital status and two tax brackets for each status, yielding four test cases.
- Test a handful of *boundary* conditions, such as an income that is at the boundary between two brackets, and a zero income.
- If you are responsible for error checking (which is discussed in Section 3.8), also test an invalid input, such as a negative income.

Each branch of your program should be covered by a test case.

Make a list of the test cases and the expected outputs:

Test Case	Expected Output	Comment
30,000 s	3,000	10% bracket
72,000 s	13,200	3,200 + 25% of 40,000
50,000 m	5,000	10% bracket
104,000 m	16,400	6,400 + 25% of 40,000
32,000 s	3,200	boundary case
0	0	boundary case

It is a good idea to design test cases before implementing a program.

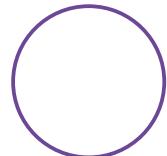
When you develop a set of test cases, it is helpful to have a flowchart of your program (see Section 3.5). Check off each branch that has a test case. Include test cases for the boundary cases of each decision. For example, if a decision checks whether an input is less than 100, test with an input of 100.

It is always a good idea to design test cases *before* starting to code. Working through the test cases gives you a better understanding of the algorithm that you are about to implement.

SELF CHECK



27. Using Figure 1 on page 85 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `ElevatorSimulation.java` program in Section 3.1.
28. What is a boundary test case for the algorithm in How To 3.1 on page 95? What is the expected output?
29. Using Figure 3 on page 99 as a guide, follow the process described in Section 3.6 to design a set of test cases for the `EarthquakeStrength.java` program in Section 3.3.
30. Suppose you are designing a part of a program for a medical robot that has a sensor returning an x - and y -location (measured in cm). You need to check whether the sensor location is inside the circle, outside the circle, or on the boundary (specifically, having a distance of less than 1 mm from the boundary). Assume the circle has center $(0, 0)$ and a radius of 2 cm. Give a set of test cases.



Practice It Now you can try these exercises at the end of the chapter: R3.16, R3.17.

Programming Tip 3.6



Make a Schedule and Make Time for Unexpected Problems

Commercial software is notorious for being delivered later than promised. For example, Microsoft originally promised that its Windows Vista operating system would be available late in 2003, then in 2005, then in March 2006; it finally was released in January 2007. Some of the early promises might not have been realistic. It was in Microsoft's interest to let prospective customers expect the imminent availability of the product. Had customers known the actual delivery date, they might have switched to a different product in the meantime. Undeniably, though, Microsoft had not anticipated the full complexity of the tasks it had set itself to solve.

Microsoft can delay the delivery of its product, but it is likely that you cannot. As a student or a programmer, you are expected to manage your time wisely and to finish your assignments on time. You can probably do simple programming exercises the night before the due date, but an assignment that looks twice as hard may well take four times as long, because more things can go wrong. You should therefore make a schedule whenever you start a programming project.

First, estimate realistically how much time it will take you to:

- Design the program logic.
- Develop test cases.
- Type the program in and fix syntax errors.
- Test and debug the program.

For example, for the income tax program I might estimate an hour for the design; 30 minutes for developing test cases; an hour for data entry and fixing syntax errors; and an hour for testing and debugging. That is a total of 3.5 hours. If I work two hours a day on this project, it will take me almost two days.

Then think of things that can go wrong. Your computer might break down. You might be stumped by a problem with the computer system. (That is a particularly important concern for beginners. It is *very* common to lose a day over a trivial problem just because it takes time to track down a person who knows the magic command to overcome it.) As a rule of thumb, *double* the time of your estimate. That is, you should start four days, not two days, before the due date. If nothing went wrong, great; you have the program done two days early. When the inevitable problem occurs, you have a cushion of time that protects you from embarrassment and failure.



Bananastock/Media Bakery

Make a schedule for your programming work and build in time for problems.

Special Topic 3.5



Logging

Sometimes you run a program and you are not sure where it spends its time. To get a printout of the program flow, you can insert trace messages into the program, such as this one:

```
if (status == SINGLE)
{
    System.out.println("status is SINGLE");
    ...
}
```

However, there is a problem with using `System.out.println` for trace messages. When you are done testing the program, you need to remove all print statements that produce trace messages. If you find another error, however, you need to stick the print statements back in.

To overcome this problem, you should use the `Logger` class, which allows you to turn off the trace messages without removing them from the program.

Instead of printing directly to `System.out`, use the global logger object that is returned by the call `Logger.getGlobal()`. (Prior to Java 7, you obtained the global logger as `Logger.getLogger("global")`.) Then call the `info` method:

```
Logger.getGlobal().info("status is SINGLE");
```

By default, the message is printed. But if you call

```
Logger.getGlobal().setLevel(Level.OFF);
```

at the beginning of the `main` method of your program, all log message printing is suppressed. Set the level to `Level.INFO` to turn logging of `info` messages on again. Thus, you can turn off the log messages when your program works fine, and you can turn them back on if you find another error. In other words, using `Logger.getGlobal().info` is just like `System.out.println`, except that you can easily activate and deactivate the logging.

The `Logger` class has many other options for industrial-strength logging. Check out the API documentation if you want to have more control over logging.

Logging messages can be deactivated when testing is complete.

3.7 Boolean Variables and Operators

The Boolean type `boolean` has two values, `false` and `true`.

Jon Patton/E+/Getty Images, Inc.



A Boolean variable is also called a flag because it can be either up (true) or down (false).

Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).

Sometimes, you need to evaluate a logical condition in one part of a program and use it elsewhere. To store a condition that can be true or false, you use a *Boolean variable*. Boolean variables are named after the mathematician George Boole (1815–1864), a pioneer in the study of logic.

In Java, the `boolean` data type has exactly two values, denoted `false` and `true`. These values are not strings or integers; they are special values, just for Boolean variables. Here is a declaration of a Boolean variable:

```
boolean failed = true;
```

You can use the value later in your program to make a decision:

```
if (failed) // Only executed if failed has been set to true
{
    ...
}
```

When you make complex decisions, you often need to combine Boolean values. An operator that combines Boolean conditions is called a **Boolean operator**. In Java, the `&&` operator (called *and*) yields `true` only when both conditions are `true`. The `||` operator (called *or*) yields the result `true` if at least one of the conditions is `true`.

Suppose you write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water. (At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.) Water is liquid if the temperature is greater than zero *and* less than 100:

```
if (temp > 0 && temp < 100) { System.out.println("Liquid"); }
```

The condition of the test has two parts, joined by the `&&` operator. Each part is a Boolean value that can be `true` or `false`. The combined expression is `true` if both individual expressions are `true`. If either one of the expressions is `false`, then the result is also `false` (see Figure 8).

The Boolean operators `&&` and `||` have a lower precedence than the relational operators. For that reason, you can write relational expressions on either side of the Boolean operators without using parentheses. For example, in the expression

```
temp > 0 && temp < 100
```

the expressions `temp > 0` and `temp < 100` are evaluated first. Then the `&&` operator combines the results. Appendix B shows a table of the Java operators and their precedence.

A	B	<code>A && B</code>	A	B	<code>A B</code>	A	<code>!A</code>
true	true	true	true	true	true	true	false
true	false	false	true	false	true	false	true
false	true	false	false	true	true		
false	false	false	false	false	false		

Figure 8 Boolean Truth Tables

At this geyser in Iceland, you can see ice, liquid water, and steam.



© toosfisStockphoto.

FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a program comparing numbers using Boolean expressions.

To invert a condition, use the `!` (*not*) operator.

Conversely, let's test whether water is *not* liquid at a given temperature. That is the case when the temperature is at most 0 *or* at least 100. Use the `||` (*or*) operator to combine the expressions:

```
if (temp <= 0 || temp >= 100) { System.out.println("Not liquid"); }
```

Figure 9 shows flowcharts for these examples.

Sometimes you need to *invert* a condition with the *not* Boolean operator. The `!` operator takes a single condition and evaluates to true if that condition is false and to false if the condition is true. In this example, output occurs if the value of the Boolean variable `frozen` is false:

```
if (!frozen) { System.out.println("Not frozen"); }
```

Table 5 illustrates additional examples of evaluating Boolean operators.

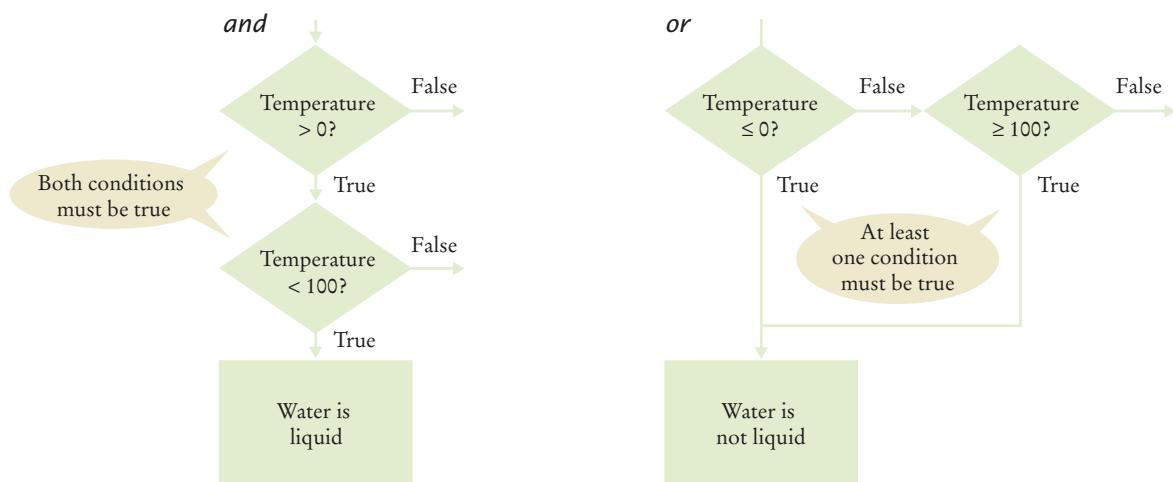


Figure 9 Flowcharts for *and* and *or* Combinations

Table 5 Boolean Operator Examples

Expression	Value	Comment
<code>0 < 200 && 200 < 100</code>	<code>false</code>	Only the first condition is true.
<code>0 < 200 200 < 100</code>	<code>true</code>	The first condition is true.
<code>0 < 200 100 < 200</code>	<code>true</code>	The <code> </code> is not a test for “either-or”. If both conditions are true, the result is true.
<code>0 < x && x < 100 x == -1</code>	<code>(0 < x && x < 100) x == -1</code>	The <code>&&</code> operator has a higher precedence than the <code> </code> operator (see Appendix B).
 <code>0 < x < 100</code>	Error	Error: This expression does not test whether <code>x</code> is between 0 and 100. The expression <code>0 < x</code> is a Boolean value. You cannot compare a Boolean value with the integer 100.
 <code>x && y > 0</code>	Error	Error: This expression does not test whether <code>x</code> and <code>y</code> are positive. The left-hand side of <code>&&</code> is an integer, <code>x</code> , and the right-hand side, <code>y > 0</code> , is a Boolean value. You cannot use <code>&&</code> with an integer argument.
<code>!(0 < 200)</code>	<code>false</code>	<code>0 < 200</code> is true, therefore its negation is <code>false</code> .
<code>frozen == true</code>	<code>frozen</code>	There is no need to compare a Boolean variable with <code>true</code> .
<code>frozen == false</code>	<code>!frozen</code>	It is clearer to use <code>!</code> than to compare with <code>false</code> .

**SELF CHECK**

31. Suppose `x` and `y` are two integers. How do you test whether both of them are zero?
32. How do you test whether at least one of them is zero?
33. How do you test whether *exactly one of them* is zero?
34. What is the value of `!!frozen`?
35. What is the advantage of using the type `boolean` rather than strings "false"/"true" or integers 0/1?

Practice It Now you can try these exercises at the end of the chapter: R3.30, E3.17, E3.18.

Common Error 3.5**Combining Multiple Relational Operators**

Consider the expression

```
if (0 <= temp <= 100) // Error
```

This looks just like the mathematical test $0 \leq \text{temp} \leq 100$. But in Java, it is a compile-time error.

Let us dissect the condition. The first half, `0 <= temp`, is a test with an outcome true or false. The outcome of that test (true or false) is then compared against 100. This seems to make no

sense. Is true larger than 100 or not? Can one compare truth values and numbers? In Java, you cannot. The Java compiler rejects this statement.

Instead, use `&&` to combine two separate tests:

```
if (0 <= temp && temp <= 100) . . .
```

Another common error, along the same lines, is to write

```
if (input == 1 || 2) . . . // Error
```

to test whether `input` is 1 or 2. Again, the Java compiler flags this construct as an error. You cannot apply the `||` operator to numbers. You need to write two Boolean expressions and join them with the `||` operator:

```
if (input == 1 || input == 2) . . .
```

Common Error 3.6



Confusing `&&` and `||` Conditions

It is a surprisingly common error to confuse *and* and *or* conditions. A value lies between 0 and 100 if it is at least 0 *and* at most 100. It lies outside that range if it is less than 0 *or* greater than 100. There is no golden rule; you just have to think carefully.

Often the *and* or *or* is clearly stated, and then it isn't too hard to implement it. But sometimes the wording isn't as explicit. It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined. Consider these instructions for filing a tax return. You can claim single filing status if any one of the following is true:

- You were never married.
- You were legally separated or divorced on the last day of the tax year.
- You were widowed, and did not remarry.

Since the test passes if *any one* of the conditions is true, you must combine the conditions with *or*. Elsewhere, the same instructions state that you may use the more advantageous status of married filing jointly if all five of the following conditions are true:

- Your spouse died less than two years ago and you did not remarry.
- You have a child whom you can claim as dependent.
- That child lived in your home for all of the tax year.
- You paid over half the cost of keeping up your home for this child.
- You filed a joint return with your spouse the year he or she died.

Because *all* of the conditions must be true for the test to pass, you must combine them with an *and*.

Special Topic 3.6



Short-Circuit Evaluation of Boolean Operators

The `&&` and `||` operators are computed using short-circuit evaluation. In other words, logical expressions are evaluated from left to right, and evaluation stops as soon as the truth value is determined. When an `&&` is evaluated and the first condition is false, the second condition is not evaluated, because it does not matter what the outcome of the second test is.

For example, consider the expression

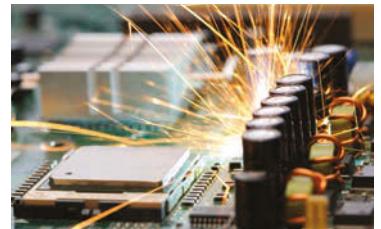
```
quantity > 0 && price / quantity < 10
```

The `&&` and `||` operators are computed using short-circuit evaluation: As soon as the truth value is determined, no further conditions are evaluated.

Suppose the value of quantity is zero. Then the test `quantity > 0` fails, and the second test is not attempted. That is just as well, because it is illegal to divide by zero.

Similarly, when the first condition of an `||` expression is true, then the remainder is not evaluated because the result must be true.

This process is called *short-circuit evaluation*.



© YourPechkin/iStockphoto

In a short circuit, electricity travels along the path of least resistance. Similarly, short-circuit evaluation takes the fastest path for computing the result of a Boolean expression.

Special Topic 3.7



De Morgan's Law

Humans generally have a hard time comprehending logical conditions with *not* operators applied to *and/or* expressions. De Morgan's Law, named after the logician Augustus De Morgan (1806–1871), can be used to simplify these Boolean expressions.

Suppose we want to charge a higher shipping rate if we don't ship within the continental United States.

```
if (!(country.equals("USA") && !state.equals("AK") && !state.equals("HI")))
{
    shippingCharge = 20.00;
}
```

This test is a little bit complicated, and you have to think carefully through the logic. When it is *not* true that the country is USA *and* the state is not Alaska *and* the state is not Hawaii, then charge \$20.00. Huh? It is not true that some people won't be confused by this code.

The computer doesn't care, but it takes human programmers to write and maintain the code. Therefore, it is useful to know how to simplify such a condition.

De Morgan's Law has two forms: one for the negation of an *and* expression and one for the negation of an *or* expression:

<code>!(A && B)</code>	is the same as	<code>!A !B</code>
<code>!(A B)</code>	is the same as	<code>!A && !B</code>

De Morgan's law tells you how to negate && and || conditions.

Pay particular attention to the fact that the *and* and *or* operators are *reversed* by moving the *not* inward. For example, the negation of "the state is Alaska *or* it is Hawaii",

`!(state.equals("AK") || state.equals("HI"))`
is "the state is not Alaska *and* it is not Hawaii":

`!state.equals("AK") && !state.equals("HI")`

Now apply the law to our shipping charge computation:

```
!(country.equals("USA")
&& !state.equals("AK")
&& !state.equals("HI"))
```

is equivalent to

```
!country.equals("USA")
|| !!state.equals("AK")
|| !!state.equals("HI")
```

Because two `!` cancel each other out, the result is the simpler test

```
!country.equals("USA")
|| state.equals("AK")
|| state.equals("HI")
```

In other words, higher shipping charges apply when the destination is outside the United States or to Alaska or Hawaii.

To simplify conditions with negations of *and* or *or* expressions, it is usually a good idea to apply De Morgan's Law to move the negations to the innermost level.

3.8 Application: Input Validation

Tetra Images/Media Bakery



Like a quality control worker, you want to make sure that user input is correct before processing it.

An important application for the `if` statement is *input validation*. Whenever your program accepts user input, you need to make sure that the user-supplied values are valid before you use them in your computations.

Consider our elevator simulation program. Assume that the elevator panel has buttons labeled 1 through 20 (but not 13). The following are illegal inputs:

- The number 13
- Zero or a negative number
- A number larger than 20
- An input that is not a sequence of digits, such as `five`

In each of these cases, we will want to give an error message and exit the program.

It is simple to guard against an input of 13:

```
if (floor == 13)
{
    System.out.println("Error: There is no thirteenth floor.");
```

Here is how you ensure that the user doesn't enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
    System.out.println("Error: The floor must be between 1 and 20.");
```

However, dealing with an input that is not a valid integer is a more serious problem. When the statement

```
floor = in.nextInt();
```

Call the `hasNextInt` or `hasNextDouble` method to ensure that the next input is a number.

is executed, and the user types in an input that is not an integer (such as `five`), then the integer variable `floor` is not set. Instead, a run-time exception occurs and the program is terminated. To avoid this problem, you should first call the `hasNextInt` method which checks whether the next input is an integer. If that method returns `true`, you can safely call `nextInt`. Otherwise, print an error message and exit the program.

```
if (in.hasNextInt())
{
    int floor = in.nextInt();
    Process the input value.
```

```

    }
else
{
    System.out.println("Error: Not an integer.");
}

```

Here is the complete elevator simulation program with input validation:

sec08/ElevatorSimulation2.java

```

1 import java.util.Scanner;
2
3 /**
4     This program simulates an elevator panel that skips the 13th floor, checking for
5     input errors.
6 */
7 public class ElevatorSimulation2
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Floor: ");
13        if (in.hasNextInt())
14        {
15            // Now we know that the user entered an integer
16
17            int floor = in.nextInt();
18
19            if (floor == 13)
20            {
21                System.out.println("Error: There is no thirteenth floor.");
22            }
23            else if (floor <= 0 || floor > 20)
24            {
25                System.out.println("Error: The floor must be between 1 and 20.");
26            }
27            else
28            {
29                // Now we know that the input is valid
30
31                int actualFloor = floor;
32                if (floor > 13)
33                {
34                    actualFloor = floor - 1;
35                }
36
37                System.out.println("The elevator will travel to the actual floor "
38                               + actualFloor);
39            }
40        }
41        else
42        {
43            System.out.println("Error: Not an integer.");
44        }
45    }
46 }

```

Program Run

```

Floor: 13
Error: There is no thirteenth floor.

```



- 36.** In the `ElevatorSimulation2` program, what is the output when the input is
- 100?
 - 1?
 - 20?
 - thirteen?

- 37.** Your task is to rewrite lines 19–26 of the `ElevatorSimulation2` program so that there is a single `if` statement with a complex condition. What is the condition?

```
if (. . .)
{
    System.out.println("Error: Invalid floor number");
}
```

- 38.** In the Sherlock Holmes story “The Adventure of the Sussex Vampire”, the inimitable detective uttered these words: “Matilda Briggs was not the name of a young woman, Watson, ... It was a ship which is associated with the giant rat of Sumatra, a story for which the world is not yet prepared.” Over a hundred years later, researchers found giant rats in Western New Guinea, another part of Indonesia.

Suppose you are charged with writing a program that processes rat weights. It contains the statements

```
System.out.print("Enter weight in kg: ");
double weight = in.nextDouble();
```

What input checks should you supply?

When processing inputs, you want to reject values that are too large. But how large is too large? These giant rats, found in Western New Guinea, are about five times the size of a city rat.

- 39.** Run the following test program and supply inputs 2 and three at the prompts. What happens? Why?

```
import java.util.Scanner
public class Test
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int m = in.nextInt();
        System.out.print("Enter another integer: ");
        int n = in.nextInt();
        System.out.println(m + " " + n);
    }
}
```

Practice It Now you can try these exercises at the end of the chapter: R3.3, R3.33, E3.12.



VIDEO EXAMPLE 3.2

The Genetic Code

Watch this Video Example to see how to build a “decoder ring” for the genetic code. Go to wiley.com/go/bj1o2videos to view Video Example 3.2.



© benjaminalbiach/
iStockphoto



Computing & Society 3.2 Artificial Intelligence

When one uses a sophisticated computer program such as a tax preparation package, one is bound to attribute some intelligence to the computer. The computer asks sensible questions and makes computations that we find a mental challenge. After all, if doing one's taxes were easy, we wouldn't need a computer to do it for us.

As programmers, however, we know that all this apparent intelligence is an illusion. Human programmers have carefully "coached" the software in all possible scenarios, and it simply replays the actions and decisions that were programmed into it.

Would it be possible to write computer programs that are genuinely intelligent in some sense? From the earliest days of computing, there was a sense that the human brain might be nothing but an immense computer, and that it might well be feasible to program computers to imitate some processes of human thought. Serious research into *artificial intelligence* began in the mid-1950s, and the first twenty years brought some impressive successes. Programs that play chess—surely an activity that appears to require remarkable intellectual powers—have become so good that they now routinely beat all but the best human players. As far back as 1975, an *expert-system* program called Mycin gained fame for being better in diagnosing meningitis in patients than the average physician.

From the very outset, one of the stated goals of the AI community was to produce software that could translate text from one language to another, for example from English to Russian. That undertaking proved to be enormously complicated. Human language appears to be much more subtle and interwoven with the human experience than had originally been thought. Systems such as Apple's Siri can answer common questions about the weather, appointments, and traffic. However, beyond a narrow range, they are more entertaining than useful.

In some areas, artificial intelligence technology has seen substantial advances. One of the most astounding examples is the rapid development of self-driving car technology. Starting in

2004, the Defense Advanced Research Projects Agency (DARPA) organized a series of competitions in which computer-controlled vehicles had to complete an obstacle course without a human driver or remote control. The first event was a disappointment, with none of the entrants finishing the route. In 2005, five vehicles completed a grueling 212 km course in the Mojave desert. Stanford's Stanley came in first, with an average speed of 30 km/h. In 2007, DARPA moved the competition to an "urban" environment, an abandoned air force base. Vehicles had to be able to interact with each other, following California traffic laws. Self-driving cars are now tested on public roads in several states, and it is expected that they will become commercially available within a decade.

When a system with artificial intelligence replaces a human in an activity such as giving medical advice or driving a vehicle, an important question arises. Who is responsible for mistakes? We accept that human doctors and drivers occasionally make mistakes with lethal consequences. Will we do the same for medical expert systems and self-driving cars?



Vaughn Youtz/Zuma Press

Winner of the 2007 DARPA Urban Challenge

CHAPTER SUMMARY

Use the if statement to implement a decision.

- The if statement allows a program to carry out different actions depending on the nature of the data to be processed.



Implement comparisons of numbers and objects.

- Use relational operators (`< <= > >= == !=`) to compare numbers.
- Do not use the `==` operator to compare strings. Use the `equals` method instead.
- The `compareTo` method compares strings in lexicographic order.

**Implement complex decisions that require multiple if statements.**

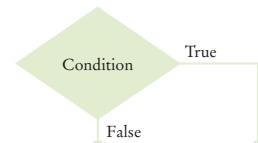
- Multiple `if` statements can be combined to evaluate complex decisions.
- When using multiple `if` statements, test general conditions after more specific conditions.

Implement decisions whose branches require further decisions.

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

Draw flowcharts for visualizing the control flow of a program.

- Flowcharts are made up of elements for tasks, input/output, and decisions.
- Each branch of a decision can contain tasks and further decisions.
- Never point an arrow inside another branch.

**Design test cases for your programs.**

- Each branch of your program should be covered by a test case.
- It is a good idea to design test cases before implementing a program.
- Logging messages can be deactivated when testing is complete.

Use the Boolean data type to store and combine conditions that can be true or false.

- The Boolean type `boolean` has two values, `false` and `true`.
- Java has two Boolean operators that combine conditions: `&&` (*and*) and `||` (*or*).
- To invert a condition, use the `!` (*not*) operator.
- The `&&` and `||` operators are computed using *short-circuit evaluation*: As soon as the truth value is determined, no further conditions are evaluated.
- De Morgan's law tells you how to negate `&&` and `||` conditions.

Apply if statements to detect whether user input is valid.

- Call the `hasNextInt` or `hasNextDouble` method to ensure that the next input is a number.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

```
java.lang.String
    equals
    compareTo
java.util.Scanner
    hasNextDouble
    hasNextInt
```

```
java.util.logging.Level
    INFO
    OFF
java.util.logging.Logger
    getGlobal
    info
    setLevel
```

REVIEW EXERCISES

- R3.1** What is the value of each variable after the if statement?

- `int n = 1; int k = 2; int r = n;`
`if (k < n) { r = k; }`
- `int n = 1; int k = 2; int r;`
`if (n < k) { r = k; }`
`else { r = k + n; }`
- `int n = 1; int k = 2; int r = k;`
`if (r < k) { n = r; }`
`else { k = n; }`
- `int n = 1; int k = 2; int r = 3;`
`if (r < n + k) { r = 2 * n; }`
`else { k = 2 * r; }`

- R3.2** Explain the difference between

```
s = 0;
if (x > 0) { s++; }
if (y > 0) { s++; }
```

and

```
s = 0;
if (x > 0) { s++; }
else if (y > 0) { s++; }
```

- R3.3** Find the errors in the following if statements.

- `if x > 0 then System.out.print(x);`
- `if (1 + x > Math.pow(x, Math.sqrt(2))) { y = y + x; }`
- `if (x = 1) { y++; }`
- `x = in.nextInt();
if (in.hasNextInt())
{
 sum = sum + x;`

```

    }
else
{
    System.out.println("Bad input for x");
}
e. String letterGrade = "F";
if (grade >= 90) { letterGrade = "A"; }
if (grade >= 80) { letterGrade = "B"; }
if (grade >= 70) { letterGrade = "C"; }
if (grade >= 60) { letterGrade = "D"; }

```

■ R3.4 What do these code fragments print?

- a. int n = 1;
 int m = -1;
 if (n < -m) { System.out.print(n); }
 else { System.out.print(m); }
- b. int n = 1;
 int m = -1;
 if (-n >= m) { System.out.print(n); }
 else { System.out.print(m); }
- c. double x = 0;
 double y = 1;
 if (Math.abs(x - y) < 1) { System.out.print(x); }
 else { System.out.print(y); }
- d. double x = Math.sqrt(2);
 double y = 2;
 if (x * x == y) { System.out.print(x); }
 else { System.out.print(y); }

■ ■ R3.5 Suppose x and y are variables of type `double`. Write a code fragment that sets y to x if x is positive and to 0 otherwise.

■ ■ R3.6 Suppose x and y are variables of type `double`. Write a code fragment that sets y to the absolute value of x without calling the `Math.abs` function. Use an `if` statement.

■ ■ R3.7 Explain why it is more difficult to compare floating-point numbers than integers. Write Java code to test whether an integer n equals 10 and whether a floating-point number x is approximately equal to 10.

■ ■ R3.8 Given two pixels on a computer screen with integer coordinates (x_1, y_1) and (x_2, y_2) , write conditions to test whether they are

- a. The same pixel.
- b. Very close together (with distance < 5).

■ R3.9 It is easy to confuse the = and == operators. Write a test program containing the statement

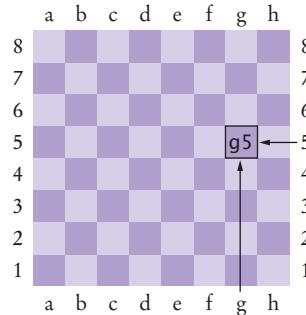
```
if (floor = 13)
```

What error message do you get? Write another test program containing the statement

```
count == 0;
```

What does your compiler do when you compile the program?

- R3.10** Each square on a chess board can be described by a letter and number, such as g5 in this example:



The following pseudocode describes an algorithm that determines whether a square with a given letter and number is dark (black) or light (white).

```

If the letter is an a, c, e, or g
  If the number is odd
    color = "black"
  Else
    color = "white"
Else
  If the number is even
    color = "black"
  Else
    color = "white"

```

Using the procedure in Programming Tip 3.5, trace this pseudocode with input g5.

- R3.11** Give a set of four test cases for the algorithm of Exercise R3.10 that covers all branches.
- R3.12** In a scheduling program, we want to check whether two appointments overlap. For simplicity, appointments start at a full hour, and we use military time (with hours 0–24). The following pseudocode describes an algorithm that determines whether the appointment with start time **start1** and end time **end1** overlaps with the appointment with start time **start2** and end time **end2**.

```

If start1 > start2
  s = start1
Else
  s = start2
If end1 < end2
  e = end1
Else
  e = end2
If s < e
  The appointments overlap.
Else
  The appointments don't overlap.

```

Trace this algorithm with an appointment from 10–12 and one from 11–13, then with an appointment from 10–11 and one from 12–13.

- R3.13 Draw a flowchart for the algorithm in Exercise R3.12.
- R3.14 Draw a flowchart for the algorithm in Exercise E3.13.
- R3.15 Draw a flowchart for the algorithm in Exercise E3.14.
- R3.16 Develop a set of test cases for the algorithm in Exercise R3.12.
- R3.17 Develop a set of test cases for the algorithm in Exercise E3.14.
- R3.18 Write pseudocode for a program that prompts the user for a month and day and prints out whether it is one of the following four holidays:
 - New Year's Day (January 1)
 - Independence Day (July 4)
 - Veterans Day (November 11)
 - Christmas Day (December 25)
- R3.19 Write pseudocode for a program that assigns letter grades for a quiz, according to the following table:

Score	Grade
90-100	A
80-89	B
70-79	C
60-69	D
< 60	F
- R3.20 Explain how the lexicographic ordering of strings in Java differs from the ordering of words in a dictionary or telephone book. *Hint:* Consider strings such as IBM, wiley.com, Century 21, and While-U-Wait.
- R3.21 Of the following pairs of strings, which comes first in lexicographic order?
 - a. "Tom", "Jerry"
 - b. "Tom", "Tomato"
 - c. "church", "Churchill"
 - d. "car manufacturer", "carburetor"
 - e. "Harry", "hairy"
 - f. "Java", "Car"
 - g. "Tom", "Tom"
 - h. "Car", "Carl"
 - i. "car", "bar"
- R3.22 Explain the difference between an if/else if/else sequence and nested if statements. Give an example of each.
- R3.23 Give an example of an if/else if/else sequence where the order of the tests does not matter. Give an example where the order of the tests matters.
- R3.24 Rewrite the condition in Section 3.3 to use < operators instead of >= operators. What is the impact on the order of the comparisons?
- R3.25 Give a set of test cases for the tax program in Exercise P3.8. Manually compute the expected results.

- **R3.26** Make up a Java code example that shows the dangling `else` problem using the following statement: A student with a GPA of at least 1.5, but less than 2, is on probation. With less than 1.5, the student is failing.
- **R3.27** Complete the following truth table by finding the truth values of the Boolean expressions for all combinations of the Boolean inputs `p`, `q`, and `r`.

<code>p</code>	<code>q</code>	<code>r</code>	<code>(p && q) !r</code>	<code>!(p && (q !r))</code>
false	false	false		
false	false	true		
false	true	false		
...				
5 more combinations				
...				

- **R3.28** True or false? $A \&\& B$ is the same as $B \&\& A$ for any Boolean conditions A and B .
- **R3.29** The “advanced search” feature of many search engines allows you to use Boolean operators for complex queries, such as “(cats OR dogs) AND NOT pets”. Contrast these search operators with the Boolean operators in Java.
- **R3.30** Suppose the value of `b` is `false` and the value of `x` is 0. What is the value of each of the following expressions?
- `b && x == 0`
 - `b || x == 0`
 - `!b && x == 0`
 - `!b || x == 0`
 - `b && x != 0`
 - `b || x != 0`
 - `!b && x != 0`
 - `!b || x != 0`
- **R3.31** Simplify the following expressions. Here, `b` is a variable of type `boolean`.
- `b == true`
 - `b == false`
 - `b != true`
 - `b != false`
- **R3.32** Simplify the following statements. Here, `b` is a variable of type `boolean` and `n` is a variable of type `int`.
- `if (n == 0) { b = true; } else { b = false; }`
(Hint: What is the value of `n == 0`?)
 - `if (n == 0) { b = false; } else { b = true; }`
 - `b = false; if (n > 1) { if (n < 2) { b = true; } }`
 - `if (n < 1) { b = true; } else { b = n > 2; }`

- **R3.33** What is wrong with the following program?

```

System.out.print("Enter the number of quarters: ");
int quarters = in.nextInt();
if (in.hasNextInt())
{
    total = total + quarters * 0.25;
    System.out.println("Total: " + total);
}
else
{
    System.out.println("Input error.");
}

```

PRACTICE EXERCISES

- **E3.1** Write a program that reads an integer and prints whether it is negative, zero, or positive.
- ■ **E3.2** Write a program that reads a floating-point number and prints “zero” if the number is zero. Otherwise, print “positive” or “negative”. Add “small” if the absolute value of the number is less than 1, or “large” if it exceeds 1,000,000.
- ■ **E3.3** Write a program that reads an integer and prints how many digits the number has, by checking whether the number is ≥ 10 , ≥ 100 , and so on. (Assume that all integers are less than ten billion.) If the number is negative, first multiply it with -1 .
- ■ **E3.4** Write a program that reads three numbers and prints “all the same” if they are all the same, “all different” if they are all different, and “neither” otherwise.
- ■ **E3.5** Write a program that reads three numbers and prints “increasing” if they are in increasing order, “decreasing” if they are in decreasing order, and “neither” otherwise. Here, “increasing” means “strictly increasing”, with each value larger than its predecessor. The sequence 3 4 4 would not be considered increasing.
- ■ **E3.6** Repeat Exercise E3.5, but before reading the numbers, ask the user whether increasing/decreasing should be “strict” or “lenient”. In lenient mode, the sequence 3 4 4 is increasing and the sequence 4 4 4 is both increasing and decreasing.
- ■ **E3.7** Write a program that reads in three integers and prints “in order” if they are sorted in ascending *or* descending order, or “not in order” otherwise. For example,

```

1 2 5   in order
1 5 2   not in order
5 2 1   in order
1 2 2   in order

```

- ■ **E3.8** Write a program that reads four integers and prints “two pairs” if the input consists of two matching pairs (in some order) and “not two pairs” otherwise. For example,

```

1 2 2 1   two pairs
1 2 2 3   not two pairs
2 2 2 2   two pairs

```

- ■ **E3.9** A compass needle points a given number of degrees away from North, measured clockwise. Write a program that reads the angle and prints out the nearest compass direction; one of N, NE, E, SE, S, SW, W, NW. In the case of a tie, prefer the nearest principal direction (N, E, S, or W).
- **E3.10** Write a program that reads a temperature value and the letter C for Celsius or F for Fahrenheit. Print whether water is liquid, solid, or gaseous at the given temperature at sea level.
- **E3.11** The boiling point of water drops by about one degree centigrade for every 300 meters (or 1,000 feet) of altitude. Improve the program of Exercise E3.10 to allow the user to supply the altitude in meters or feet.
- **E3.12** Add error handling to Exercise E3.11. If the user does not enter a number when expected, or provides an invalid unit for the altitude, print an error message and end the program.
- ■ **E3.13** When two points in time are compared, each given as hours (in military time, ranging from 0 and 23) and minutes, the following pseudocode determines which comes first.

```

If hour1 < hour2
    time1 comes first.
Else if hour1 and hour2 are the same
    If minute1 < minute2
        time1 comes first.
    Else if minute1 and minute2 are the same
        time1 and time2 are the same.
    Else
        time2 comes first.
Else
    time2 comes first.

```

Write a program that prompts the user for two points in time and prints the time that comes first, then the other time.

- ■ **E3.14** The following algorithm yields the season (Spring, Summer, Fall, or Winter) for a given month and day.

```

If month is 1, 2, or 3, season = "Winter"
Else if month is 4, 5, or 6, season = "Spring"
Else if month is 7, 8, or 9, season = "Summer"
Else if month is 10, 11, or 12, season = "Fall"
If month is divisible by 3 and day >= 21
    If season is "Winter", season = "Spring"
    Else if season is "Spring", season = "Summer"
    Else if season is "Summer", season = "Fall"
    Else season = "Winter"

```

Write a program that prompts the user for a month and day and then prints the season, as determined by this algorithm.



© rotofrank/iStockphoto.

- E3.15** Write a program that reads in two floating-point numbers and tests whether they are the same up to two decimal places. Here are two sample runs.

```
Enter two floating-point numbers: 2.0 1.99998
They are the same up to two decimal places.
Enter two floating-point numbers: 2.0 1.98999
They are different.
```

- E3.16** *Unit conversion.* Write a unit conversion program that asks the users from which unit they want to convert (fl. oz, gal, oz, lb, in, ft, mi) and to which unit they want to convert (ml, l, g, kg, mm, cm, m, km). Reject incompatible conversions (such as gal → km). Ask for the value to be converted, then display the result:

```
Convert from? gal
Convert to? ml
Value? 2.5
2.5 gal = 9462.5 ml
```

- E3.17** Write a program that prompts the user to provide a single character from the alphabet. Print Vowel or Consonant, depending on the user input. If the user input is not a letter (between a and z or A and Z), or is a string of length > 1, print an error message.

- E3.18** Write a program that asks the user to enter a month (1 for January, 2 for February, and so on) and then prints the number of days in the month. For February, print “28 or 29 days”.

```
Enter a month: 5
30 days
```

Do not use a separate if/else branch for each month. Use Boolean operators.

PROGRAMMING PROJECTS

- P3.1** Write a program that translates a letter grade into a number grade. Letter grades are A, B, C, D, and F, possibly followed by + or -. Their numeric values are 4, 3, 2, 1, and 0. There is no F+ or F-. A + increases the numeric value by 0.3, a - decreases it by 0.3. However, an A+ has value 4.0.

```
Enter a letter grade: B-
The numeric value is 2.7.
```

- P3.2** Write a program that translates a number between 0 and 4 into the closest letter grade. For example, the number 2.8 (which might have been the average of several grades) would be converted to B-. Break ties in favor of the better grade; for example 2.85 should be a B.

- P3.3** Write a program that takes user input describing a playing card in the following shorthand notation:

A	Ace
2 ... 10	Card values
J	Jack
Q	Queen

K	King
D	Diamonds
H	Hearts
S	Spades
C	Clubs

Your program should print the full description of the card. For example,

```
Enter the card notation: QS
Queen of Spades
```

- P3.4 Write a program that reads in three floating-point numbers and prints the largest of the three inputs. For example:

```
Please enter three numbers: 4 9 2.5
The largest number is 9.
```

- P3.5 Write a program that reads in three strings and sorts them lexicographically.

```
Enter three strings: Charlie Able Baker
Able
Baker
Charlie
```

- P3.6 Write a program that prompts for the day and month of the user's birthday and then prints a horoscope. Make up fortunes for programmers, like this:

```
Please enter your birthday (month and day): 6 16
Gemini are experts at figuring out the behavior of complicated programs.
You feel where bugs are coming from and then stay one step ahead. Tonight,
your style wins approval from a tough critic.
```

Each fortune should contain the name of the astrological sign. (You will find the names and date ranges of the signs at a distressingly large number of sites on the Internet.)



- P3.7 The original U.S. income tax of 1913 was quite simple. The tax was

- 1 percent on the first \$50,000.
- 2 percent on the amount over \$50,000 up to \$75,000.
- 3 percent on the amount over \$75,000 up to \$100,000.
- 4 percent on the amount over \$100,000 up to \$250,000.
- 5 percent on the amount over \$250,000 up to \$500,000.
- 6 percent on the amount over \$500,000.

There was no separate schedule for single or married taxpayers. Write a program that computes the income tax according to this schedule.

- **P3.8** Write a program that computes taxes for the following schedule.

If your status is Single and if the taxable income is over	but not over	the tax is	of the amount over
\$0	\$8,000	10%	\$0
\$8,000	\$32,000	\$800 + 15%	\$8,000
\$32,000		\$4,400 + 25%	\$32,000
If your status is Married and if the taxable income is over	but not over	the tax is	of the amount over
\$0	\$16,000	10%	\$0
\$16,000	\$64,000	\$1,600 + 15%	\$16,000
\$64,000		\$8,800 + 25%	\$64,000

■■■ **P3.9** The `TaxCalculator.java` program uses a simplified version of the 2008 U.S. income tax schedule. Look up the tax brackets and rates for the current year, for both single and married filers, and implement a program that computes the actual income tax.

■ **P3.10** Write a program that reads in the x - and y -coordinates of two corner points of a rectangle and then prints out whether the rectangle is a square, or is in “portrait” or “landscape” orientation.

■■■ **P3.11** Write a program that reads in the x - and y -coordinates of three corner points of a triangle and prints out whether it has an obtuse angle, a right angle, or only acute angles.

■■■ **P3.12** Write a program that reads in the x - and y -coordinates of four corner points of a quadrilateral and prints out whether it is a square, a rectangle, a trapezoid, a rhombus, or none of those shapes.

■■■ **P3.13** *Roman numbers.* Write a program that converts a positive integer into the Roman number system. The Roman number system has digits

I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

Numbers are formed according to the following rules:

- Only numbers up to 3,999 are represented.
- As in the decimal system, the thousands, hundreds, tens, and ones are expressed separately.

- c. The numbers 1 to 9 are expressed as

I	1
II	2
III	3
IV	4
V	5
VI	6
VII	7
VIII	8
IX	9



© Straitshooter/iStockphoto.

As you can see, an I preceding a V or X is subtracted from the value, and you can never have more than three I's in a row.

- d. Tens and hundreds are done the same way, except that the letters X, L, C and C, D, M are used instead of I, V, X, respectively.

Your program should take an input, such as 1978, and convert it to Roman numerals, MCMLXXVIII.

- P3.14 A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the *Gregorian correction* applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000). Write a program that asks the user for a year and computes whether that year is a leap year. Use a single if statement and Boolean operators.

- P3.15 French country names are feminine when they end with the letter e, masculine otherwise, except for the following which are masculine even though they end with e:

- le Belize
- le Cambodge
- le Mexique
- le Mozambique
- le Zaïre
- le Zimbabwe

Write a program that reads the French name of a country and adds the article: le for masculine or la for feminine, such as le Canada or la Belgique.

However, if the country name starts with a vowel, use l'; for example, l'Afghanistan.

For the following plural country names, use les:

- les Etats-Unis
- les Pays-Bas

- Business P3.16 Write a program to simulate a bank transaction. There are two bank accounts: checking and savings. First, ask for the initial balances of the bank accounts; reject negative balances. Then ask for the transactions; options are deposit, withdrawal, and transfer. Then ask for the account; options are checking and savings. Then ask for the amount; reject transactions that overdraw an account. At the end, print the balances of both accounts.

■ Business P3.17 Write a program that reads in the name and salary of an employee. Here the salary will denote an *hourly* wage, such as \$9.25. Then ask how many hours the employee worked in the past week. Be sure to accept fractional hours. Compute the pay. Any overtime work (over 40 hours per week) is paid at 150 percent of the regular wage. Print a paycheck for the employee.

■ Business P3.18 When you use an automated teller machine (ATM) with your bank card, you need to use a personal identification number (PIN) to access your account. If a user fails more than three times when entering the PIN, the machine will block the card. Assume that the user's PIN is "1234" and write a program that asks the user for the PIN no more than three times, and does the following:



© Mark Evans/iStockphoto

- If the user enters the right number, print a message saying, "Your PIN is correct", and end the program.
- If the user enters a wrong number, print a message saying, "Your PIN is incorrect" and, if you have asked for the PIN less than three times, ask for it again.
- If the user enters a wrong number three times, print a message saying "Your bank card is blocked" and end the program.

■ Business P3.19 Calculating the tip when you go to a restaurant is not difficult, but your restaurant wants to suggest a tip according to the service diners receive. Write a program that calculates a tip according to the diner's satisfaction as follows:

- Ask for the diners' satisfaction level using these ratings: 1 = Totally satisfied, 2 = Satisfied, 3 = Dissatisfied.
- If the diner is totally satisfied, calculate a 20 percent tip.
- If the diner is satisfied, calculate a 15 percent tip.
- If the diner is dissatisfied, calculate a 10 percent tip.
- Report the satisfaction level and tip in dollars and cents.

■ Business P3.20 A supermarket awards coupons depending on how much a customer spends on groceries. For example, if you spend \$50, you will get a coupon worth eight percent of that amount. The following table shows the percent used to calculate the coupon awarded for different amounts spent. Write a program that calculates and prints the value of the coupon a person can receive based on groceries purchased.

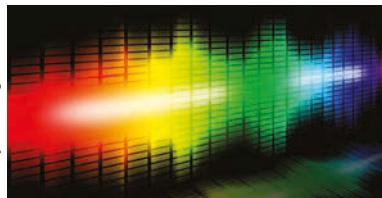
Here is a sample run:

```
Please enter the cost of your groceries: 14
You win a discount coupon of $ 1.12. (8% of your purchase)
```

Money Spent	Coupon Percentage
Less than \$10	No coupon
From \$10 to \$60	8%
More than \$60 to \$150	10%
More than \$150 to \$210	12%
More than \$210	14%

- **Science P3.21** Write a program that prompts the user for a wavelength value and prints a description of the corresponding part of the electromagnetic spectrum, as given in the following table.

© drxy/iStockphoto.



Electromagnetic Spectrum		
Type	Wavelength (m)	Frequency (Hz)
Radio Waves	$> 10^{-1}$	$< 3 \times 10^9$
Microwaves	10^{-3} to 10^{-1}	3×10^9 to 3×10^{11}
Infrared	7×10^{-7} to 10^{-3}	3×10^{11} to 4×10^{14}
Visible light	4×10^{-7} to 7×10^{-7}	4×10^{14} to 7.5×10^{14}
Ultraviolet	10^{-8} to 4×10^{-7}	7.5×10^{14} to 3×10^{16}
X-rays	10^{-11} to 10^{-8}	3×10^{16} to 3×10^{19}
Gamma rays	$< 10^{-11}$	$> 3 \times 10^{19}$

- **Science P3.22** Repeat Exercise P3.21, modifying the program so that it prompts for the frequency instead.

- ■ **Science P3.23** Repeat Exercise P3.21, modifying the program so that it first asks the user whether the input will be a wavelength or a frequency.

- ■ ■ **Science P3.24** A minivan has two sliding doors. Each door can be opened by either a dashboard switch, its inside handle, or its outside handle. However, the inside handles do not work if a child lock switch is activated. In order for the sliding doors to open, the gear shift must be in park, *and* the master unlock switch must be activated. (This book's author is the long-suffering owner of just such a vehicle.)

Your task is to simulate a portion of the control software for the vehicle. The input is a sequence of values for the switches and the gear shift, in the following order:

- Dashboard switches for left and right sliding door, child lock, and master unlock (0 for off or 1 for activated)
- Inside and outside handles on the left and right sliding doors (0 or 1)
- The gear shift setting (one of P N D 1 2 3 R).

A typical input would be 0 0 0 1 0 1 0 0 P.

Print “left door opens” and/or “right door opens” as appropriate. If neither door opens, print “both doors stay closed”.

- **Science P3.25** Sound level L in units of decibel (dB) is determined by

$$L = 20 \log_{10}(p/p_0)$$

where p is the sound pressure of the sound (in Pascals, abbreviated Pa), and p_0 is a reference sound pressure equal to 20×10^{-6} Pa (where L is 0 dB).



© nano/iStockphoto

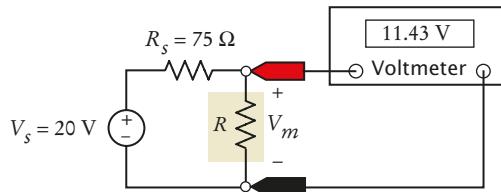


The following table gives descriptions for certain sound levels.

Threshold of pain	130 dB
Possible hearing damage	120 dB
Jack hammer at 1 m	100 dB
Traffic on a busy roadway at 10 m	90 dB
Normal conversation	60 dB
Calm library	30 dB
Light leaf rustling	0 dB

Write a program that reads a value and a unit, either dB or Pa, and then prints the closest description from the list above.

- Science P3.26 The electric circuit shown below is designed to measure the temperature of the gas in a chamber.



The resistor R represents a temperature sensor enclosed in the chamber. The resistance R , in Ω , is related to the temperature T , in $^{\circ}\text{C}$, by the equation

$$R = R_0 + kT$$

In this device, assume $R_0 = 100 \Omega$ and $k = 0.5$. The voltmeter displays the value of the voltage, V_m , across the sensor. This voltage V_m indicates the temperature, T , of the gas according to the equation

$$T = \frac{R}{k} - \frac{R_0}{k} = \frac{R_s}{k} \frac{V_m}{V_s - V_m} - \frac{R_0}{k}$$

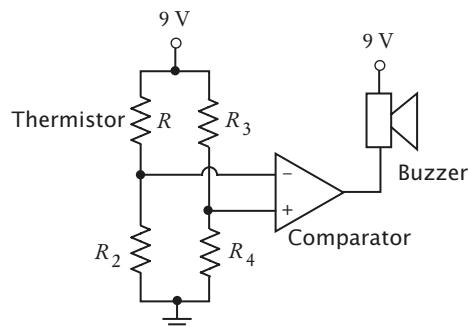
Suppose the voltmeter voltage is constrained to the range $V_{\min} = 12$ volts $\leq V_m \leq V_{\max} = 18$ volts. Write a program that accepts a value of V_m and checks that it's between 12 and 18. The program should return the gas temperature in degrees Celsius when V_m is between 12 and 18 and an error message when it isn't.

- Science P3.27 Crop damage due to frost is one of the many risks confronting farmers. The figure below shows a simple alarm circuit designed to warn of frost. The alarm circuit uses a device called a thermistor to sound a buzzer when the temperature drops below freezing. Thermistors are semiconductor devices that exhibit a temperature dependent resistance described by the equation

$$R = R_0 e^{\beta \left(\frac{1}{T} - \frac{1}{T_0} \right)}$$



where R is the resistance, in Ω , at the temperature T , in $^{\circ}\text{K}$, and R_0 is the resistance, in Ω , at the temperature T_0 , in $^{\circ}\text{K}$. β is a constant that depends on the material used to make the thermistor.



The circuit is designed so that the alarm will sound when

$$\frac{R_2}{R + R_2} < \frac{R_4}{R_3 + R_4}$$

The thermistor used in the alarm circuit has $R_0 = 33,192 \Omega$ at $T_0 = 40^\circ\text{C}$, and $\beta = 3,310 \text{ }^\circ\text{K}$. (Notice that β has units of $\text{ }^\circ\text{K}$. The temperature in $\text{ }^\circ\text{K}$ is obtained by adding 273° to the temperature in $^\circ\text{C}$.) The resistors R_2 , R_3 , and R_4 have a resistance of $156.3 \text{ k}\Omega = 156,300 \Omega$.

Write a Java program that prompts the user for a temperature in $^\circ\text{F}$ and prints a message indicating whether or not the alarm will sound at that temperature.

- Science P3.28** A mass $m = 2$ kilograms is attached to the end of a rope of length $r = 3$ meters. The mass is whirled around at high speed. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program that accepts a rotation speed v and determines whether such a speed will cause the rope to break. *Hint: $T = mv^2/r$.*

A mass m is attached to the end of a rope of length $r = 3$ meters. The rope can only be whirled around at speeds of 1, 10, 20, or 40 meters per second. The rope can withstand a maximum tension of $T = 60$ Newtons. Write a program where the user enters the value of the mass m , and the program determines the greatest speed at which it can be whirled without breaking the rope. *Hint: $T = mv^2/r$.*

- ■ Science P3.29** The average person can jump off the ground with a velocity of 7 mph without fear of leaving the planet. However, if an astronaut jumps with this velocity while standing on Halley's Comet, will the astronaut ever come back down? Create a program that allows the user to input a launch velocity (in mph) from the surface of Halley's Comet and determine whether a jumper will return to the surface. If not, the program should calculate how much more massive the comet must be in order to return the jumper to the surface.



Courtesy NASA/JPL-Caltech

Hint: Escape velocity is $v_{\text{escape}} = \sqrt{2 \frac{GM}{R}}$, where $G = 6.67 \times 10^{-11} \text{ N m}^2/\text{kg}^2$ is the gravitational constant, $M = 1.3 \times 10^{22} \text{ kg}$ is the mass of Halley's comet, and $R = 1.153 \times 10^6 \text{ m}$ is its radius.

ANSWERS TO SELF-CHECK QUESTIONS

1. Change the if statement to

```
if (floor > 14)
{
    actualFloor = floor - 2;
}
```

2. 85. 90. 85.

3. The only difference is if originalPrice is 100. The statement in Self Check 2 sets discountedPrice to 90; this one sets it to 80.

4. 95. 100. 95.

```
5. if (fuelAmount < 0.10 * fuelCapacity)
{
    System.out.println("red");
}
else
{
    System.out.println("green");
}
```

6. (a) and (b) are both true, (c) is false.

7. floor <= 13

8. The values should be compared with ==, not =.

9. input.equals("Y")

10. str.equals("") or str.length() == 0

```
11. if (scoreA > scoreB)
{
    System.out.println("A won");
}
else if (scoreA < scoreB)
{
    System.out.println("B won");
}
else
{
    System.out.println("Game tied");
}
```

```
12. if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
else { s = 0; }
```

13. You could first set s to one of the three values:

```
s = 0;
if (x > 0) { s = 1; }
else if (x < 0) { s = -1; }
```

14. The if (price <= 100) can be omitted (leaving just else), making it clear that the else branch is the sole alternative.

15. No destruction of buildings.

16. Add a branch before the final else:

```
else if (richter < 0)
{
    System.out.println("Error: Negative input");
}
```

17. 3200.

18. No. Then the computation is $0.10 \times 32000 + 0.25 \times (32000 - 32000)$.

19. No. Their individual tax is \$5,200 each, and if they married, they would pay \$10,400. Actually, taxpayers in higher tax brackets (which our program does not model) may pay higher taxes when they marry, a phenomenon known as the *marriage penalty*.

20. Change else in line 41 to

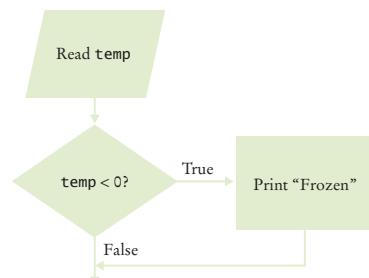
```
else if (maritalStatus.equals("m"))
```

and add another branch after line 52:

```
else
{
    System.out.println(
        "Error: marital status should be s or m.");
}
```

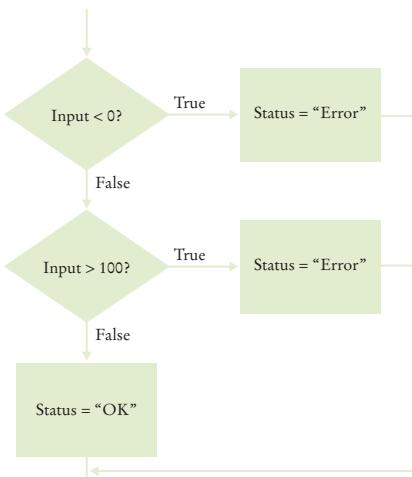
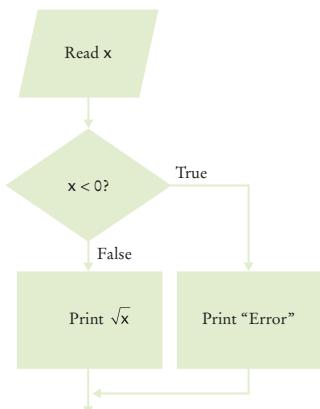
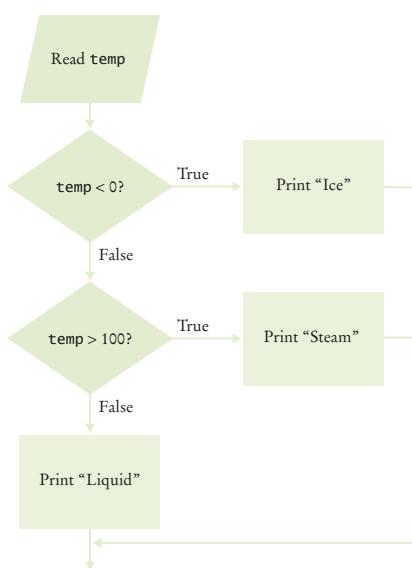
21. The higher tax rate is only applied on the income in the higher bracket. Suppose you are single and make \$31,900. Should you try to get a \$200 raise? Absolutely: you get to keep 90 percent of the first \$100 and 75 percent of the next \$100.

- 22.



23. The “True” arrow from the first decision points into the “True” branch of the second decision, creating spaghetti code.

- 24.** Here is one solution. In Section 3.7, you will see how you can combine the conditions for a more elegant solution.

**25.****26.****27.**

Test Case	Expected Output	Comment
12	12	Below 13th floor
14	13	Above 13th floor
13	?	The specification is not clear—See Section 3.8 for a version of this program with error handling

- 28.** A boundary test case is a price of \$128. A 16 percent discount should apply because the problem statement states that the larger discount applies if the price is *at least* \$128. Thus, the expected output is \$107.52.

29.

Test Case	Expected Output	Comment
9	Most structures fall	
7.5	Many buildings destroyed	
6.5	Many buildings ...	
5	Damage to poorly...	
3	No destruction...	
8.0	Most structures fall	Boundary case. In this program, boundary cases are not as significant because the behavior of an earthquake changes gradually.
-1		The specification is not clear—see Self Check 16 for a version of this program with error handling.

30.

Test Case	Expected Output	Comment
(0.5, 0.5)	inside	
(4, 2)	outside	
(0, 2)	on the boundary	Exactly on the boundary
(1.414, 1.414)	on the boundary	Close to the boundary
(0, 1.9)	inside	Not less than 1 mm from the boundary
(0, 2.1)	outside	Not less than 1 mm from the boundary

31. $x == 0 \&\& y == 0$ **32.** $x == 0 \mid\mid y == 0$ **33.** $(x == 0 \&\& y != 0) \mid\mid (y == 0 \&\& x != 0)$ **34.** The same as the value of frozen.

35. You are guaranteed that there are no other values. With strings or integers, you would need to check that no values such as "maybe" or -1 enter your calculations.

- 36.** (a) Error: The floor must be between 1 and 20.
 (b) Error: The floor must be between 1 and 20.
 (c) 19 (d) Error: Not an integer.
- 37.** `floor == 13 || floor <= 0 || floor > 20`
- 38.** Check for `in.hasNextDouble()`, to make sure a researcher didn't supply an input such as oh my. Check for `weight <= 0`, because any rat must surely have a positive weight. We don't know how giant a rat could be, but the New Guinea rats weighed no more than 2 kg. A regular house rat (*rattus rattus*) weighs up to 0.2 kg, so we'll say that any weight > 10 kg was surely an input error, perhaps confusing grams and kilograms. Thus, the checks are

```

if (in.hasNextDouble())
{
    double weight = in.nextDouble();
    if (weight <= 0)
    {
        System.out.println(
            "Error: Weight must be positive.");
    }
    else if (weight > 10)
    {
        System.out.println(
            "Error: Weight > 10 kg.");
    }
    else
    {
        Process valid weight.
    }
}
else
{
    System.out.print("Error: Not a number");
}
  
```

- 39.** The second input fails, and the program terminates without printing anything.

WORKED EXAMPLE 3.1

Extracting the Middle



Problem Statement Your task is to extract a string containing the middle character from a given string str. For example, if the string is “crate”, the result is the string “a”. However, if the string has an even number of letters, extract the middle two characters. If the string is “crates”, the result is “at”.

Step 1 Decide on the branching condition.

We need to take different actions for strings of odd and even length. Therefore, the condition is

Is the length of the string odd?

In Java, you use the remainder of division by 2 to find out whether a value is even or odd. Then the test becomes

```
if (str.length() % 2 == 1)
```

Step 2 Give pseudocode for the work that needs to be done when the condition is true.

We need to find the position of the middle character. If the length is 5, the position is 2.

c	r	a	t	e
0	1	2	3	4

In general,

```
position = str.length() / 2 (with the remainder discarded)
result = str.substring(position, position + 1)
```

Step 3 Give pseudocode for the work (if any) that needs to be done when the condition is *not* true.

Again, we need to find the position of the middle character. If the length is 6, the starting position is 2, and the ending position is 3. That is, we would call

```
result = str.substring(2, 4);
```

(Recall that the second parameter of the substring method is the first position that we do not extract.)

c	r	a	t	e	s
0	1	2	3	4	5

In general,

```
position = str.length() / 2 - 1
result = str.substring(position, position + 2)
```

Step 4 Double-check relational operators.

Do we really want `str.length() % 2 == 1`? For example, when the length is 5, $5 \% 2$ is the remainder of the division $5 / 2$, which is 1. In general, dividing an odd number by 2 leaves a remainder of 1. (Actually, dividing a negative odd number by 2 leaves a remainder of -1 , but the string length is never negative.) Therefore, our condition is correct.

Step 5 Remove duplication.

Here is the statement that we have developed:

```
If the length of str is odd
    position = str.length() / 2 (with remainder discarded)
    result = str.substring(position, position + 1)
Else
    position = str.length() / 2 - 1
    result = str.substring(position, position + 2)
```

The second statement in each branch is almost identical, but the length of the substring differs. Let's set the length in each branch:

```
If str.length() % 2 == 1
    position = str.length() / 2 (with remainder discarded)
    length = 1
Else
    position = str.length() / 2 - 1
    length = 2
    result = str.substring(position, position + length)
```

Step 6 Test both branches.

We will use a different set of strings for testing. For an odd-length string, consider "monitor". We get

```
position = str.length() / 2 = 7 / 2 = 3 (with remainder discarded)
length = 1
result = str.substring(3, 4) = "i"
```

For the even-length string "monitors", we get

```
position = str.length() / 2 - 1 = 8 / 2 - 1 = 3 (with remainder discarded)
length = 2
result = str.substring(3, 5) = "it"
```

Step 7 Assemble the if statement in Java.

Here's the completed code segment.

```
if (str.length() % 2 == 1)
{
    position = str.length() / 2;
    length = 1;
}
else
{
    position = str.length() / 2 - 1;
    length = 2;
}
String result = str.substring(position, position + length);
```

You can find the complete program in the ch03/worked_example_1 directory of the book's companion code.

LOOPS

CHAPTER GOALS

- To implement while, for, and do loops
- To hand-trace the execution of a program
- To become familiar with common loop algorithms
- To understand nested loops
- To implement programs that read and process data sets
- To use a computer for simulations



© photo75/iStockphoto.

CHAPTER CONTENTS

4.1 THE WHILE LOOP 142

- SYN** while Statement 143
- CE1** Don't Think "Are We There Yet?" 146
- CE2** Infinite Loops 147
- CE3** Off-by-One Errors 147
- C&S** The First Bug 148

4.2 PROBLEM SOLVING: HAND-TRACING 149

4.3 THE FOR LOOP 152

- SYN** for Statement 154
- PT1** Use for Loops for Their Intended Purpose Only 157
- PT2** Choose Loop Bounds That Match Your Task 157
- PT3** Count Iterations 158

4.4 THE DO LOOP 158

- PT4** Flowcharts for Loops 159

4.5 APPLICATION: PROCESSING SENTINEL VALUES 160

- ST1** The Loop-and-a-Half Problem and the break Statement 162

ST2 Redirection of Input and Output 163

- VE1** Evaluating a Cell Phone Plan

4.6 PROBLEM SOLVING: STORYBOARDS 164

4.7 COMMON LOOP ALGORITHMS 167

- HT1** Writing a Loop 171
- WE1** Credit Card Processing

4.8 NESTED LOOPS 174

- WE2** Manipulating the Pixels in an Image

4.9 PROBLEM SOLVING: SOLVE A SIMPLER PROBLEM FIRST 178

4.10 APPLICATION: RANDOM NUMBERS AND SIMULATIONS 182

- ST3** Drawing Graphical Shapes 186
- VE2** Drawing a Spiral
- C&S** Digital Piracy 188



© photo75/iStockphoto.

In a loop, a part of a program is repeated over and over, until a specific goal is reached. Loops are important for calculations that require repeated steps and for processing input consisting of many data items. In this chapter, you will learn about loop statements in Java, as well as techniques for writing programs that process input and simulate activities in the real world.

4.1 The while Loop

In this section, you will learn about *loop statements* that repeatedly execute instructions until a goal has been reached.

Recall the investment problem from Chapter 1. You put \$10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original investment?

In Chapter 1 we developed the following algorithm for this problem:

Set year to 0, balance to 10000.

year	interest	balance
0		10000

While the balance is less than \$20,000

Add 1 to the year.

Set interest to balance \times 0.05 (i.e., 5 percent interest).

Add the interest to the balance.

Report year as the answer.

You now know how to declare and update the variables in Java. What you don't yet know is how to carry out the steps "While the balance is less than \$20,000".



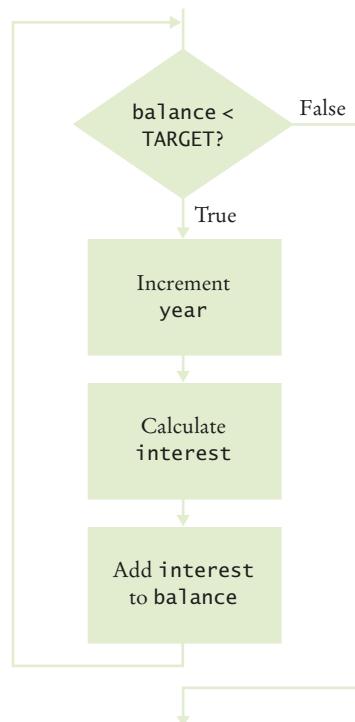
© AlterYourReality/
iStockphoto.

Because the interest earned also earns interest, a bank balance grows exponentially.

In a particle accelerator, subatomic particles traverse a loop-shaped tunnel multiple times, gaining the speed required for physical experiments. Similarly, in computer science, statements in a loop are executed while a condition is true.



© mmac72/iStockphoto.

Figure 1 Flowchart of a while Loop

A loop executes instructions repeatedly while a condition is true.

In Java, the `while` statement implements such a repetition (see Syntax 4.1). It has the form

```
while (condition)
{
    statements
}
```

As long as the condition remains true, the statements inside the `while` statement are executed. These statements are called the **body** of the `while` statement.

In our case, we want to increment the year counter and add interest while the balance is less than the target balance of \$20,000:

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

A `while` statement is an example of a **loop**. If you draw a flowchart, the flow of execution loops again to the point where the condition is tested (see Figure 1).

Syntax 4.1 while Statement

Syntax

```
while (condition)
{
    statements
}
```

This variable is declared outside the loop and updated in the loop.

If the condition never becomes false, an infinite loop occurs.
See Common Error 4.2.

This variable is created in each loop iteration.

```
double balance = 0;
.
.
.
while (balance < TARGET)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Beware of "off-by-one" errors in the loop condition.
See Common Error 4.3.

Don't put a semicolon here!
See Common Error 3.1.

These statements are executed while the condition is true.

Lining up braces is a good idea.
See Programming Tip 3.1.

Braces are not required if the body contains a single statement, but it's good to always use them.
See Programming Tip 3.2.

When you declare a variable *inside* the loop body, the variable is created for each iteration of the loop and removed after the end of each iteration. For example, consider the `interest` variable in this loop:

```
while (balance < TARGET)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
} // interest no longer declared here
```

A new interest variable
is created in each iteration.

In contrast, the `balance` and `year` variables were declared outside the loop body. That way, the same variable is used for all iterations of the loop.

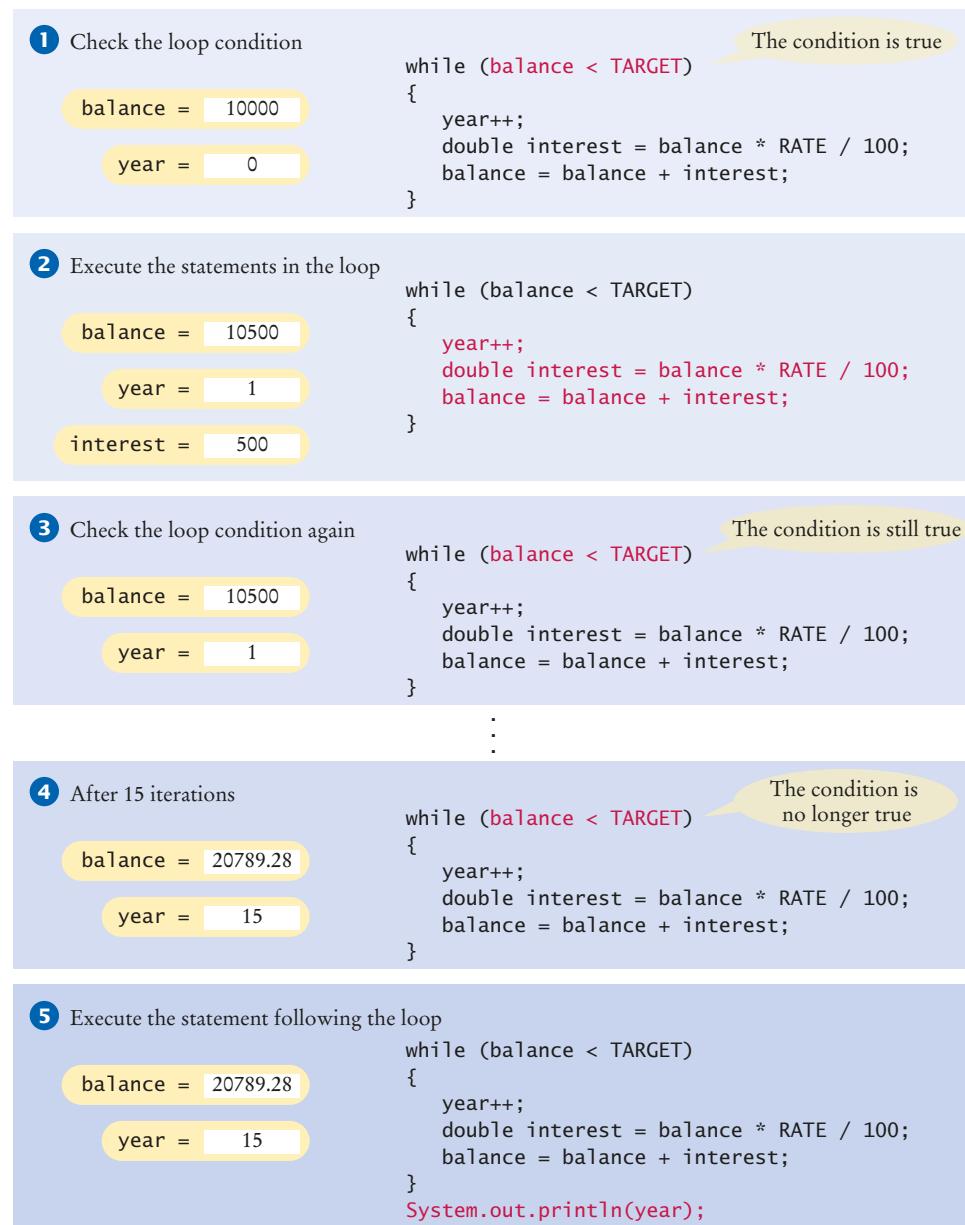


Figure 2
Execution of the
`DoubleInvestment`
Loop

Here is the program that solves the investment problem. Figure 2 illustrates the program's execution.

sec01/DoubleInvestment.java

```

1  /**
2   * This program computes the time required to double an investment.
3  */
4  public class DoubleInvestment
5  {
6      public static void main(String[] args)
7      {
8          final double RATE = 5;
9          final double INITIAL_BALANCE = 10000;
10         final double TARGET = 2 * INITIAL_BALANCE;
11
12         double balance = INITIAL_BALANCE;
13         int year = 0;
14
15         // Count the years required for the investment to double
16
17         while (balance < TARGET)
18         {
19             year++;
20             double interest = balance * RATE / 100;
21             balance = balance + interest;
22         }
23
24         System.out.println("The investment doubled after "
25                         + year + " years.");
26     }
27 }
```

Program Run

The investment doubled after 15 years.

SELF CHECK



- How many years does it take for the investment to triple? Modify the DoubleInvestment.java program and run it.
- If the interest rate is 10 percent per year, how many years does it take for the investment to double? Modify the program and run it.
- Modify the program so that the balance after each year is printed. How did you do that?
- Suppose we change the program so that the condition of the while loop is

```
while (balance <= TARGET)
```

What is the effect on the program? Why?

- What does the following loop print?

```

int n = 1;
while (n < 100)
{
    n = 2 * n;
    System.out.print(n + " ");
}
```

Practice It Now you can try these exercises at the end of the chapter: R4.4, R4.8, E4.17.

Table 1 while Loop Examples

Loop	Output	Explanation
<pre>i = 0; sum = 0; while (sum < 10) { i++; sum = sum + i; Print i and sum; }</pre>	1 1 2 3 3 6 4 10	When <code>sum</code> is 10, the loop condition is false, and the loop ends.
<pre>i = 0; sum = 0; while (sum < 10) { i++; sum = sum - i; Print i and sum; }</pre>	1 -1 2 -3 3 -6 4 -10 . . .	Because <code>sum</code> never reaches 10, this is an “infinite loop” (see Common Error 4.2 on page 147).
<pre>i = 0; sum = 0; while (sum < 0) { i++; sum = sum - i; Print i and sum; }</pre>	(No output)	The statement <code>sum < 0</code> is false when the condition is first checked, and the loop is never executed.
<pre>i = 0; sum = 0; while (sum >= 10) { i++; sum = sum + i; Print i and sum; }</pre>	(No output)	The programmer probably thought, “Stop when the sum is at least 10.” However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.1 on page 146).
<pre>i = 0; sum = 0; while (sum < 10) ; { i++; sum = sum + i; Print i and sum; }</pre>	(No output, program does not terminate)	Note the semicolon before the <code>{</code> . This loop has an empty body. It runs forever, checking whether <code>sum < 0</code> and doing nothing in the body.

Common Error 4.1**Don't Think "Are We There Yet?"**

When doing something repetitive, most of us want to know when we are done. For example, you may think, “I want to get at least \$20,000,” and set the loop condition to

```
balance >= TARGET
```

But the `while` loop thinks the opposite: How long am I allowed to keep going? The correct loop condition is

```
while (balance < TARGET)
```

In other words: “Keep at it while the balance is less than the target.”

When writing a loop condition, don't ask, “Are we there yet?” The condition determines how long the loop will keep going.



© MsSponge/iStockphoto.

Common Error 4.2**Infinite Loops**

A very annoying loop error is an *infinite loop*: a loop that runs forever and can be stopped only by killing the program or restarting the computer. If there are output statements in the program, then reams and reams of output flash by on the screen. Otherwise, the program just sits there and *hangs*, seeming to do nothing. On some systems, you can kill a hanging program by hitting Ctrl + C. On others, you can close the window in which the program runs.

A common reason for infinite loops is forgetting to update the variable that controls the loop:

```
int year = 1;
while (year <= 20)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Here the programmer forgot to add a `year++` command in the loop. As a result, the year always stays at 1, and the loop never comes to an end.

Another common reason for an infinite loop is accidentally incrementing a counter that should be decremented (or vice versa). Consider this example:

```
int year = 20;
while (year > 0)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
    year++;
}
```

The `year` variable really should have been decremented, not incremented. This is a common error because incrementing counters is so much more common than decrementing that your fingers may type the `++` on autopilot. As a consequence, `year` is always larger than 0, and the loop never ends. (Actually, `year` may eventually exceed the largest representable positive integer and *wrap around* to a negative number. Then the loop ends—of course, with a completely wrong result.)



© ohiophoto/iStockphoto.

Like this hamster who can't stop running in the treadmill, an infinite loop never ends.

Common Error 4.3**Off-by-One Errors**

Consider our computation of the number of years that are required to double an investment:

```
int year = 0;
while (balance < TARGET)
{
    year++;
    balance = balance * (1 + RATE / 100);
}
System.out.println("The investment doubled after "
+ year + " years.");
```

Should `year` start at 0 or at 1? Should you test for `balance < TARGET` or for `balance <= TARGET`? It is easy to be *off by one* in these expressions.

Some people try to solve **off-by-one errors** by randomly inserting +1 or -1 until the program seems to work—a terrible strategy. It can take a long time to compile and test all the various possibilities. Expending a small amount of mental effort is a real time saver.

Fortunately, off-by-one errors are easy to avoid, simply by thinking through a couple of test cases and using the information from the test cases to come up with a rationale for your decisions.

Should year start at 0 or at 1? Look at a scenario with simple values: an initial balance of \$100 and an interest rate of 50 percent. After year 1, the balance is \$150, and after year 2 it is \$225, or over \$200. So the investment doubled after 2 years. The loop executed two times, incrementing year each time. Hence year must start at 0, not at 1.

An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

year	balance
0	\$100
1	\$150
2	\$225

In other words, the balance variable denotes the balance after the end of the year. At the outset, the balance variable contains the balance after year 0 and not after year 1.

Next, should you use a < or <= comparison in the test? This is harder to figure out, because it is rare for the balance to be exactly twice the initial balance. There is one case when this happens, namely when the interest is 100 percent. The loop executes once. Now year is 1, and balance is exactly equal to $2 * \text{INITIAL_BALANCE}$. Has the investment doubled after one year? It has. Therefore, the loop should not execute again. If the test condition is balance < TARGET, the loop stops, as it should. If the test condition had been balance <= TARGET, the loop would have executed once more.

In other words, you keep adding interest while the balance *has not yet doubled*.



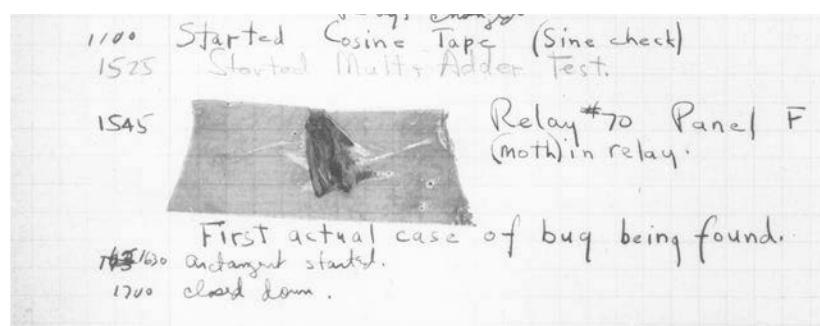
Computing & Society 4.1 The First Bug

According to legend, the first bug was found in the Mark II, a huge electromechanical computer at Harvard University. It really was caused by a bug—a moth was trapped in a relay switch.

Actually, from the note that the operator left in the log book next to the moth (see the photo), it appears as if the term “bug” had already been in active use at the time.

The pioneering computer scientist Maurice Wilkes wrote, “Somehow, at the Moore School and afterwards, one had always assumed there would be no particular difficulty in getting pro-

grams right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs.”



The First Bug

Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988. NHHC Collection.

4.2 Problem Solving: Hand-Tracing

Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.

In Programming Tip 3.5, you learned about the method of hand-tracing. When you hand-trace code or pseudocode, you write the names of the variables on a sheet of paper, mentally execute each step of the code and update the variables.

It is best to have the code written or printed on a sheet of paper. Use a marker, such as a paper clip, to mark the current line. Whenever a variable changes, cross out the old value and write the new value below. When a program produces output, also write down the output in another column.

Consider this example. What value is displayed?

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

There are three variables: `n`, `sum`, and `digit`.

<code>n</code>	<code>sum</code>	<code>digit</code>

The first two variables are initialized with 1729 and 0 before the loop is entered.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	

Because `n` is greater than zero, enter the loop. The variable `digit` is set to 9 (the remainder of dividing 1729 by 10). The variable `sum` is set to $0 + 9 = 9$.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
	9	9

Finally, `n` becomes 172. (Recall that the remainder in the division $1729 / 10$ is discarded because both arguments are integers.)

Cross out the old values and write the new ones under the old ones.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

Now check the loop condition again.

```
int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

Because `n` is still greater than zero, repeat the loop. Now `digit` becomes 2, `sum` is set to $9 + 2 = 11$, and `n` is set to 17.

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
172	9	9

Repeat the loop once again, setting `digit` to 7, `sum` to $11 + 7 = 18$, and `n` to 1.

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
172	9	9
17	11	2

Enter the loop for one last time. Now `digit` is set to 1, `sum` to 19, and `n` becomes zero.

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
172	9	9
17	11	2
1	18	7
0	19	1

```

int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);

```

Because n equals zero,
this condition is not true.

The condition `n > 0` is now false. Continue with the statement after the loop.

```

int n = 1729;
int sum = 0;
while (n > 0)
{
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);

```

n	sum	digit	output
1729	0		
172	9	9	
17	11	2	
1	18	7	
0	19	1	19

This statement is an output statement. The value that is output is the value of `sum`, which is 19.

Of course, you can get the same answer by just running the code. However, hand-tracing can give you an *insight* that you would not get if you simply ran the code. Consider again what happens in each iteration:

- We extract the last digit of `n`.
- We add that digit to `sum`.
- We strip the digit off `n`.

Hand-tracing can help you understand how an unfamiliar algorithm works.

Hand-tracing can show errors in code or pseudocode.

In other words, the loop forms the sum of the digits in `n`. You now know what the loop does for any value of `n`, not just the one in the example. (Why would anyone want to form the sum of the digits? Operations of this kind are useful for checking the validity of credit card numbers and other forms of ID numbers—see Exercise P4.21.)

Hand-tracing does not just help you understand code that works correctly. It is a powerful technique for finding errors in your code. When a program behaves in a way that you don't expect, get out a sheet of paper and track the values of the variables as you mentally step through the code.

You don't need a working program to do hand-tracing. You can hand-trace pseudocode. In fact, it is an excellent idea to hand-trace your pseudocode before you go to the trouble of translating it into actual code, to confirm that it works correctly.

SELF CHECK



6. Hand-trace the following code, showing the value of `n` and the output.

```

int n = 5;
while (n >= 0)
{
    n--;
    System.out.print(n);
}

```

7. Hand-trace the following code, showing the value of `n` and the output. What potential error do you notice?

```
int n = 1;
while (n <= 3)
{
    System.out.print(n + ", ");
    n++;
}
```

8. Hand-trace the following code, assuming that `a` is 2 and `n` is 4. Then explain what the code does for arbitrary values of `a` and `n`.

```
int r = 1;
int i = 1;
while (i <= n)
{
    r = r * a;
    i++;
}
```

9. Trace the following code. What error do you observe?

```
int n = 1;
while (n != 50)
{
    System.out.println(n);
    n = n + 10;
}
```

10. The following pseudocode is intended to count the number of digits in the number `n`:

```
count = 1
temp = n
while (temp > 10)
    Increment count.
    Divide temp by 10.0.
```

Trace the pseudocode for `n` = 123 and `n` = 100. What error do you find?

Practice It Now you can try these exercises at the end of the chapter: R4.6, R4.9.

4.3 The for Loop

The `for` loop is used when a variable runs from a starting point to an ending point with a constant increment or decrement.

It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
int counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
    System.out.println(counter);
    counter++; // Update the counter
}
```

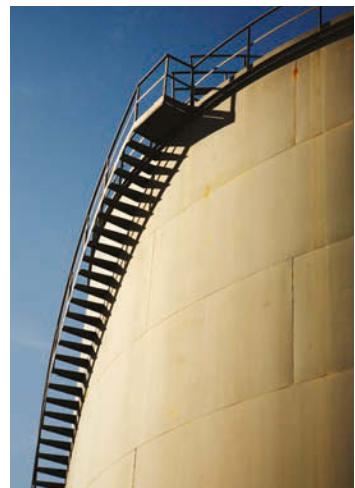
Because this loop type is so common, there is a special form for it, called the `for` loop (see Syntax 4.2).

```
for (int counter = 1; counter <= 10; counter++)
{
    System.out.println(counter);
}
```

Some people call this loop *count-controlled*. In contrast, the while loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times; ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.

The for loop neatly groups the initialization, condition, and update expressions together. However, it is important to realize that these expressions are not executed together (see Figure 3).

- The initialization is executed once, before the loop is entered. ①
- The condition is checked before each iteration. ② ⑤
- The update is executed after each iteration. ④



© Enrico Fianchini/iStockphoto.

You can visualize the for loop as an orderly sequence of steps.

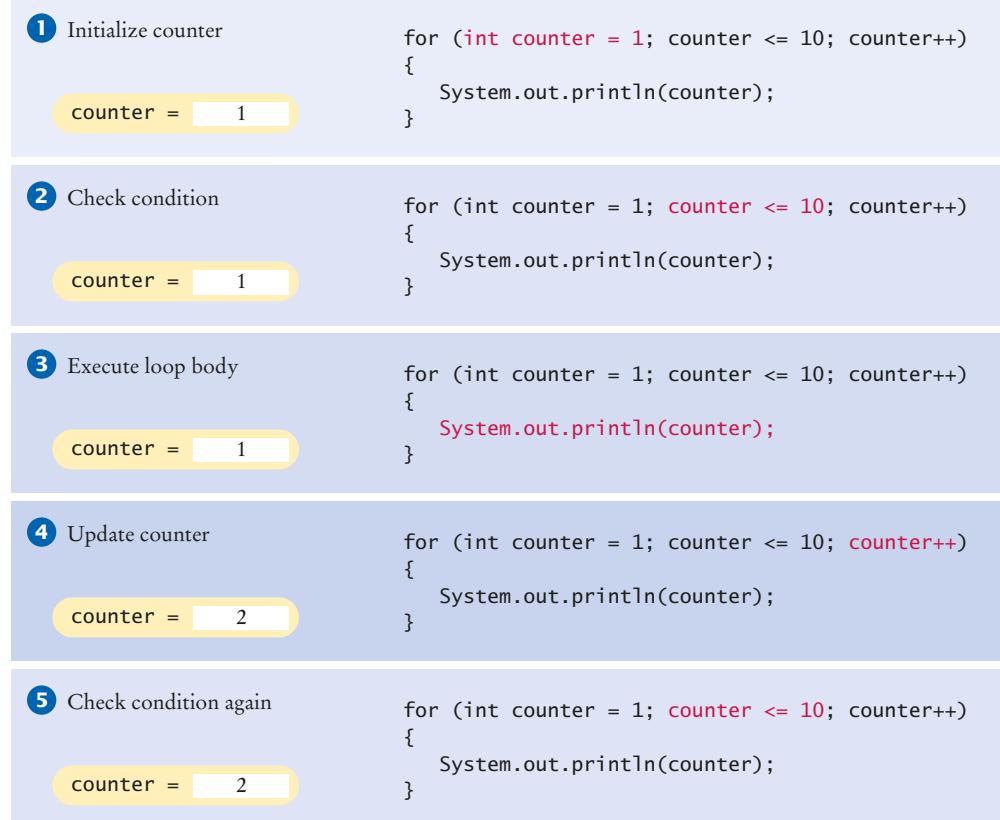


Figure 3
Execution of a
for Loop

Syntax 4.2 for Statement

```
Syntax    for (initialization; condition; update)
{  
    statements
}
```

These three
expressions should be related.
See Programming Tip 4.1.

This initialization
happens once
before the loop starts.

The condition is
checked before
each iteration.

This update is
executed after
each iteration.

The variable *i*
is defined only in this
for loop.

```
for (int i = 5; i <= 10; i++)
{
    sum = sum + i;
}
```

This loop executes 6 times.
See Programming Tip 4.3.

A for loop can count down instead of up:

```
for (int counter = 10; counter >= 0; counter--) . . .
```

The increment or decrement need not be in steps of 1:

```
for (int counter = 0; counter <= 10; counter = counter + 2) . . .
```

See Table 2 for additional variations.

So far, we have always declared the counter variable in the loop initialization:

```
for (int counter = 1; counter <= 10; counter++)
{
    . .
}
// counter no longer declared here
```

Table 2 for Loop Examples

Loop	Values of <i>i</i>	Comment
<code>for (i = 0; i <= 5; i++)</code>	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 4.3 on page 158.)
<code>for (i = 5; i >= 0; i--)</code>	5 4 3 2 1 0	Use <i>i--</i> for decreasing values.
<code>for (i = 0; i < 9; i = i + 2)</code>	0 2 4 6 8	Use <i>i = i + 2</i> for a step size of 2.
<code>for (i = 0; i != 9; i = i + 2)</code>	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use <i><</i> or <i><=</i> instead of <i>!=</i> to avoid this problem.
<code>for (i = 1; i <= 20; i = i * 2)</code>	1 2 4 8 16	You can specify any rule for modifying <i>i</i> , such as doubling it in every step.
<code>for (i = 0; i < str.length(); i++)</code>	0 1 2 ... until the last valid index of the string <i>str</i>	In the loop body, use the expression <code>str.charAt(i)</code> to get the <i>i</i> th character.

Such a variable is declared for all iterations of the loop, but you cannot use it after the loop. If you declare the counter variable before the loop, you can continue to use it after the loop:

```
int counter;
for (counter = 1; counter <= 10; counter++)
{
    ...
}
// counter still declared here
```

Here is a typical use of the for loop. We want to print the balance of our savings account over a period of years, as shown in this table:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The for loop pattern applies because the variable year starts at 1 and then moves in constant increments until it reaches the target:

```
for (int year = 1; year <= nyears; year++)
{
    Update balance.
    Print year and balance.
}
```

Following is the complete program. Figure 4 shows the corresponding flowchart.

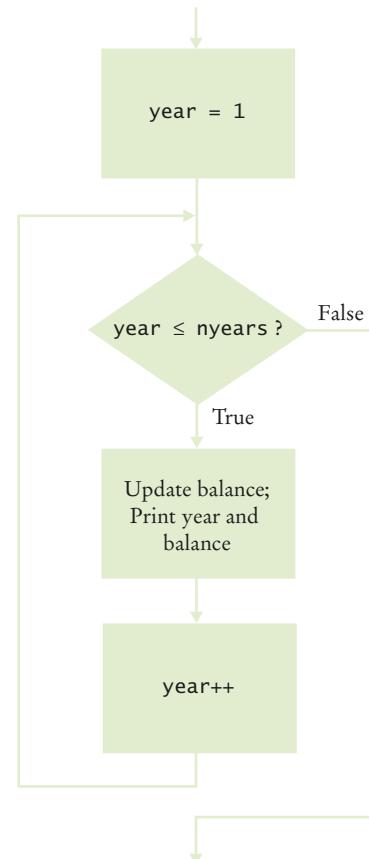


Figure 4 Flowchart of a for Loop

sec03/InvestmentTable.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program prints a table showing the growth of an investment.
5 */
6 public class InvestmentTable
7 {
8     public static void main(String[] args)
9     {
10         final double RATE = 5;
11         final double INITIAL_BALANCE = 10000;
12     }
13 }
```

```

12     double balance = INITIAL_BALANCE;
13
14     System.out.print("Enter number of years: ");
15     Scanner in = new Scanner(System.in);
16     int nyears = in.nextInt();
17
18     // Print the table of balances for each year
19
20     for (int year = 1; year <= nyears; year++)
21     {
22         double interest = balance * RATE / 100;
23         balance = balance + interest;
24         System.out.printf("%4d %10.2f\n", year, balance);
25     }
26 }
27 }
```

Program Run

```

Enter number of years: 10
1 10500.00
2 11025.00
3 11576.25
4 12155.06
5 12762.82
6 13400.96
7 14071.00
8 14774.55
9 15513.28
10 16288.95
```

Another common use of the `for` loop is to traverse all characters of a string:

```

for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    Process ch.
}
```

Note that the counter variable `i` starts at 0, and the loop is terminated when `i` reaches the length of the string. For example, if `str` has length 5, `i` takes on the values 0, 1, 2, 3, and 4. These are the valid positions in the string.

SELF CHECK



11. Write the `for` loop of the `InvestmentTable.java` program as a `while` loop.
 12. How many numbers does this loop print?
- ```

for (int n = 10; n >= 0; n--)
{
 System.out.println(n);
}
```
13. Write a `for` loop that prints all even numbers between 10 and 20 (inclusive).
  14. Write a `for` loop that computes the sum of the integers from 1 to `n`.
  15. How would you modify the `for` loop of the `InvestmentTable.java` program to print all balances until the investment has doubled?

**Practice It** Now you can try these exercises at the end of the chapter: R4.7, R4.13, E4.8, E4.16.

**Programming Tip 4.1****Use for Loops for Their Intended Purpose Only**

A for loop is an *idiom* for a loop of a particular form. A value runs from the start to the end, with a constant increment or decrement.

The compiler won't check whether the initialization, condition, and update expressions are related. For example, the following loop is legal:

```
// Confusing—unrelated expressions
for (System.out.print("Inputs: "); in.hasNextDouble(); sum = sum + x)
{
 x = in.nextDouble();
}
```

However, programmers reading such a for loop will be confused because it does not match their expectations. Use a while loop for iterations that do not follow the for idiom.

You should also be careful not to update the loop counter in the body of a for loop. Consider the following example:

```
for (int counter = 1; counter <= 100; counter++)
{
 if (counter % 10 == 0) // Skip values that are divisible by 10
 {
 counter++; // Bad style—you should not update the counter in a for loop
 }
 System.out.println(counter);
}
```

Updating the counter inside a for loop is confusing because the counter is updated *again* at the end of the loop iteration. In some loop iterations, counter is incremented once, in others twice. This goes against the intuition of a programmer who sees a for loop.

If you find yourself in this situation, you can either change from a for loop to a while loop, or implement the “skipping” behavior in another way. For example:

```
for (int counter = 1; counter <= 100; counter++)
{
 if (counter % 10 != 0) // Skip values that are divisible by 10
 {
 System.out.println(counter);
 }
}
```

**Programming Tip 4.2****Choose Loop Bounds That Match Your Task**

Suppose you want to print line numbers that go from 1 to 10. Of course, you will use a loop:

```
for (int i = 1; i <= 10; i++)
```

The values for i are bounded by the relation  $1 \leq i \leq 10$ . Because there are  $\leq$  on both bounds, the bounds are called **symmetric**.

When traversing the characters in a string, it is more natural to use the bounds

```
for (int i = 0; i < str.length(); i++)
```

In this loop, i traverses all valid positions in the string. You can access the  $i$ th character as `str.charAt(i)`. The values for i are bounded by  $0 \leq i < str.length()$ , with a  $\leq$  to the left and a  $<$  to the right. That is appropriate, because `str.length()` is not a valid position. Such bounds are called **asymmetric**.

In this case, it is not a good idea to use symmetric bounds:

```
for (int i = 0; i <= str.length() - 1; i++) // Use < instead
```

The asymmetric form is easier to understand.

## Programming Tip 4.3

**Count Iterations**

Finding the correct lower and upper bounds for an iteration can be confusing. Should you start at 0 or at 1? Should you use `<= b` or `< b` as a termination condition?

Counting the number of iterations is a very useful device for better understanding a loop. Counting is easier for loops with asymmetric bounds. The loop

```
for (int i = a; i < b; i++)
```

is executed  $b - a$  times. For example, the loop traversing the characters in a string,

```
for (int i = 0; i < str.length(); i++)
```

runs `str.length()` times. That makes perfect sense, because there are `str.length()` characters in a string.

The loop with symmetric bounds,

```
for (int i = a; i <= b; i++)
```

is executed  $b - a + 1$  times. That “+1” is the source of many programming errors.

For example,

```
for (int i = 0; i <= 10; i++)
```

runs 11 times. Maybe that is what you want; if not, start at 1 or use `< 10`.

One way to visualize this “+1” error is by looking at a fence. Each section has one fence post to the left, and there is a final post on the right of the last section. Forgetting to count the last value is often called a “fence post error”.



*How many posts do you need for a fence with four sections? It is easy to be “off by one” with problems such as this one.*

© akaplummer/iStockphoto.

## 4.4 The do Loop

The do loop is appropriate when the loop body must be executed at least once.

Sometimes you want to execute the body of a loop at least once and perform the loop test after the body is executed. The do loop serves that purpose:

```
do
{
 statements
}
while (condition);
```

The body of the do loop is executed first, then the condition is tested.

Some people call such a loop a *post-test loop* because the condition is tested after completing the loop body. In contrast, while and for loops are *pre-test loops*. In those loop types, the condition is tested before entering the loop body.

A typical example for a do loop is input validation. Suppose you ask a user to enter a value  $< 100$ . If the user doesn’t pay attention and enters a larger value, you ask again, until the value is correct. Of course, you cannot test the value until the user has entered it. This is a perfect fit for the do loop (see Figure 5):

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj1o2code](http://wiley.com/go/bj1o2code) to download a program to illustrate the use of the do loop for input validation.

**Figure 5** Flowchart of a do Loop

```
int value;
do
{
 System.out.print("Enter an integer < 100: ");
 value = in.nextInt();
}
while (value >= 100);
```

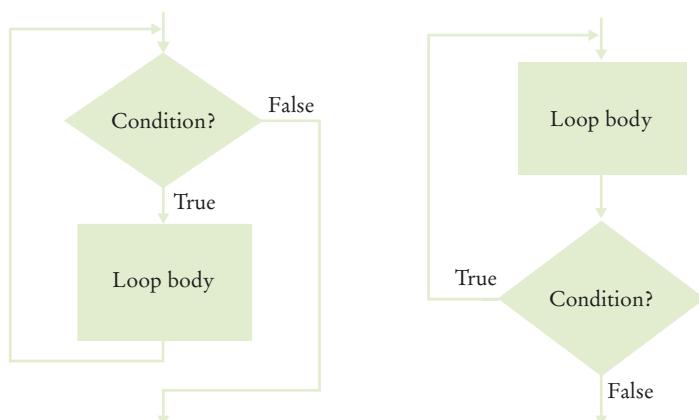
**SELF CHECK**

- 16.** Suppose that we want to check for inputs that are at least 0 and at most 100. Modify the do loop in this section for this check.
- 17.** Rewrite the input check do loop using a while loop. What is the disadvantage of your solution?
- 18.** Suppose Java didn't have a do loop. Could you rewrite any do loop as a while loop?
- 19.** Write a do loop that reads integers and computes their sum. Stop when reading the value 0.
- 20.** Write a do loop that reads integers and computes their sum. Stop when reading a zero or the same value twice in a row. For example, if the input is 1 2 3 4 4, then the sum is 14 and the loop stops.

**Practice It** Now you can try these exercises at the end of the chapter: R4.12, R4.19, R4.20.

**Programming Tip 4.4****Flowcharts for Loops**

In Section 3.5, you learned how to use flowcharts to visualize the flow of control in a program. There are two types of loops that you can include in a flowchart; they correspond to a while loop and a do loop in Java. They differ in the placement of the condition—either before or after the loop body.



As described in Section 3.5, you want to avoid “spaghetti code” in your flowcharts. For loops, that means that you never want to have an arrow that points inside a loop body.

## 4.5 Application: Processing Sentinel Values

A sentinel value denotes the end of a data set, but it is not part of the data.

In this section, you will learn how to write loops that read and process a sequence of input values.

Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence. Sometimes you are lucky and no input value can be zero. Then you can prompt the user to keep entering numbers, or 0 to finish the sequence. If zero is allowed but negative numbers are not, you can use -1 to indicate termination.

Such a value, which is not an actual input, but serves as a signal for termination, is called a **sentinel**.

Let's put this technique to work in a program that computes the average of a set of salary values. In our sample program, we will use -1 as a sentinel. An employee would surely not work for a negative salary, but there may be volunteers who work for free.

Inside the loop, we read an input. If the input is not -1, we process it. In order to compute the average, we need the total sum of all salaries, and the number of inputs.

```
salary = in.nextDouble();
if (salary != -1)
{
 sum = sum + salary;
 count++;
}
```

We stay in the loop while the sentinel value is not detected.

```
while (salary != -1)
{
 .
 .
}
```

There is just one problem: When the loop is entered for the first time, no data value has been read. We must make sure to initialize salary with some value other than the sentinel:

```
double salary = 0;
// Any value other than -1 will do
```

After the loop has finished, we compute and print the average. Here is the complete program:

### **sec05/SentinelDemo.java**

```
1 import java.util.Scanner;
2 /**
3 * This program prints the average of salary values that are terminated with a sentinel.
4 */
5
```



© Rhoberazzi/iStockphoto.

*In the military, a sentinel guards a border or passage. In computer science, a sentinel value denotes the end of an input sequence or the border between input sequences.*

```

6 public class SentinelDemo
7 {
8 public static void main(String[] args)
9 {
10 double sum = 0;
11 int count = 0;
12 double salary = 0;
13 System.out.print("Enter salaries, -1 to finish: ");
14 Scanner in = new Scanner(System.in);
15
16 // Process data until the sentinel is entered
17
18 while (salary != -1)
19 {
20 salary = in.nextDouble();
21 if (salary != -1)
22 {
23 sum = sum + salary;
24 count++;
25 }
26 }
27
28 // Compute and print the average
29
30 if (count > 0)
31 {
32 double average = sum / count;
33 System.out.println("Average salary: " + average);
34 }
35 else
36 {
37 System.out.println("No data");
38 }
39 }
40 }
```

### Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

You can use a Boolean variable to control a loop. Set the variable before entering the loop, then set it to the opposite to leave the loop.

Some programmers don't like the "trick" of initializing the input variable with a value other than the sentinel. Another approach is to use a Boolean variable:

```

System.out.print("Enter salaries, -1 to finish: ");
boolean done = false;
while (!done)
{
 value = in.nextDouble();
 if (value == -1)
 {
 done = true;
 }
 else
 {
 Process value.
 }
}
```

Special Topic 4.1 on page 162 shows an alternative mechanism for leaving such a loop.

Now consider the case in which any number (positive, negative, or zero) can be an acceptable input. In such a situation, you must use a sentinel that is not a number (such as the letter Q). As you have seen in Section 3.8, the condition

```
in.hasNextDouble()
```

is false if the next input is not a floating-point number. Therefore, you can read and process a set of inputs with the following loop:

```
System.out.print("Enter values, Q to quit: ");
while (in.hasNextDouble())
{
 value = in.nextDouble();
 Process value.
}
```

### SELF CHECK



21. What does the `SentinelDemo.java` program print when the user immediately types -1 when prompted for a value?
22. Why does the `SentinelDemo.java` program have *two* checks of the form `salary != -1`
23. What would happen if the declaration of the `salary` variable in `SentinelDemo.java` was changed to  
`double salary = -1;`
24. In the last example of this section, we prompt the user “Enter values, Q to quit.” What happens when the user enters a different letter?
25. What is wrong with the following loop for reading a sequence of values?

```
System.out.print("Enter values, Q to quit: ");
do
{
 double value = in.nextDouble();
 sum = sum + value;
 count++;
}
while (in.hasNextDouble());
```

**Practice It** Now you can try these exercises at the end of the chapter: R4.16, P4.16, P4.17.

### Special Topic 4.1

#### The Loop-and-a-Half Problem and the break Statement

Consider again this loop for processing inputs until a sentinel value has been reached:

```
boolean done = false;
while (!done)
{
 double value = in.nextDouble();
 if (value == -1)
 {
 done = true;
 }
 else
 {
 Process value.
 }
}
```



The actual test for loop termination is in the middle of the loop, not at the top. This is called a **loop and a half** because one must go halfway into the loop before knowing whether one needs to terminate.

As an alternative, you can use the `break` reserved word.

```
while (true)
{
 double value = in.nextDouble();
 if (value == -1) { break; }
 Process value.
}
```

The `break` statement breaks out of the enclosing loop, independent of the loop condition. When the `break` statement is encountered, the loop is terminated, and the statement following the loop is executed.

In the loop-and-a-half case, `break` statements can be beneficial. But it is difficult to lay down clear rules as to when they are safe and when they should be avoided. We do not use the `break` statement in this book.

## Special Topic 4.2



### Redirection of Input and Output

Consider the `SentinelDemo` program that computes the average value of an input sequence. If you use such a program, then it is quite likely that you already have the values in a file, and it seems a shame that you have to type them all in again. The command line interface of your operating system provides a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. If you type

```
java SentinelDemo < numbers.txt
```

the program is executed, but it no longer expects input from the keyboard. All input commands get their input from the file `numbers.txt`. This process is called *input redirection*.

Input redirection is an excellent tool for testing programs. When you develop a program and fix its bugs, it is boring to keep entering the same input every time you run the program. Spend a few minutes putting the inputs into a file, and use redirection.

You can also redirect output. In this program, that is not terribly useful. If you run

```
java SentinelDemo < numbers.txt > output.txt
```

the file `output.txt` contains the input prompts and the output, such as

```
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Average salary: 15
```

However, redirecting output is obviously useful for programs that produce lots of output. You can format or print the file containing the output.

Use input redirection to read input from a file.  
Use output redirection to capture program output in a file.



## VIDEO EXAMPLE 4.1

### Evaluating a Cell Phone Plan

In this Video Example, you will learn how to design a program that computes the cost of a cell phone plan from actual usage data. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 4.1.



## 4.6 Problem Solving: Storyboards

A storyboard consists of annotated sketches for each step in an action sequence.

Developing a storyboard helps you understand the inputs and outputs that are required for a program.

When you design a program that interacts with a user, you need to make a plan for that interaction. What information does the user provide, and in which order? What information will your program display, and in which format? What should happen when there is an error? When does the program quit?

This planning is similar to the development of a movie or a computer game, where *storyboards* are used to plan action sequences. A storyboard is made up of panels that show a sketch of each step. Annotations explain what is happening and note any special situations. Storyboards are also used to develop software—see Figure 6.

Making a storyboard is very helpful when you begin designing a program. You need to ask yourself which information you need in order to compute the answers that the program user wants. You need to decide how to present those answers. These are important considerations that you want to settle before you design an algorithm for computing the answers.

Let's look at a simple example. We want to write a program that helps users with questions such as "How many tablespoons are in a pint?" or "How many inches are 30 centimeters?"

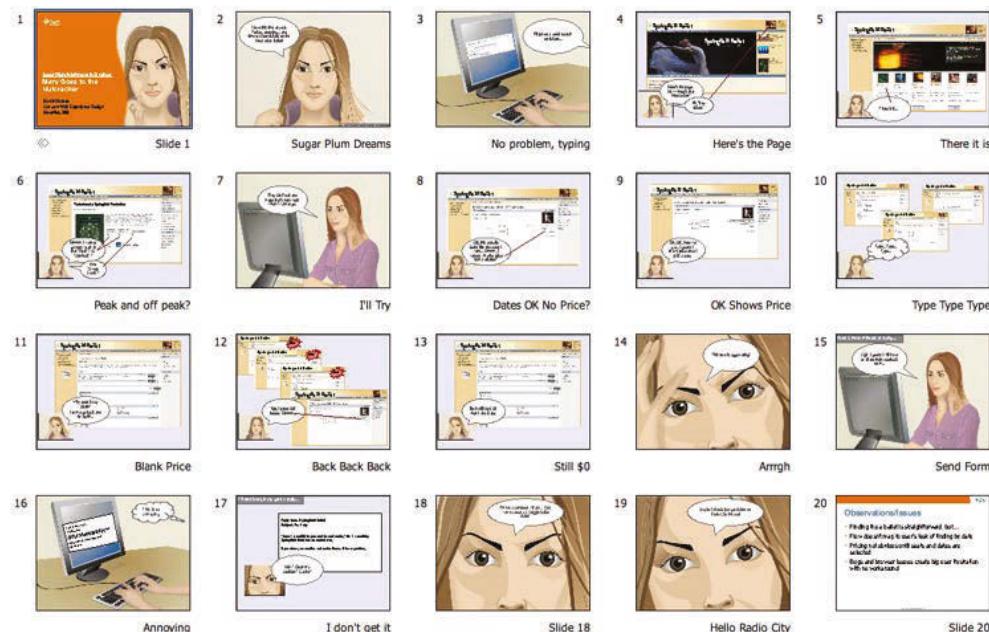
What information does the user provide?

- The quantity and unit to convert from
- The unit to convert to

What if there is more than one quantity? A user may have a whole table of centimeter values that should be converted into inches.

What if the user enters units that our program doesn't know how to handle, such as ångström?

What if the user asks for impossible conversions, such as inches to gallons?



Courtesy of Martin Hardee.

**Figure 6**  
Storyboard for the Design of a Web Application

Let's get started with a storyboard panel. It is a good idea to write the user inputs in a different color. (Underline them if you don't have a color pen handy.)

### Converting a Sequence of Values

What unit do you want to convert from? **cm**

What unit do you want to convert to? **in**

Enter values, terminated by zero Allows conversion of multiple values

**30**

**30 cm = 11.81 in** Format makes clear what got converted

**100**

**100 cm = 39.37 in**

**0**

What unit do you want to convert from?

The storyboard shows how we deal with a potential confusion. A user who wants to know how many inches are 30 centimeters may not read the first prompt carefully and specify inches. But then the output is "30 in = 76.2 cm", alerting the user to the problem.

The storyboard also raises an issue. How is the user supposed to know that "cm" and "in" are valid units? Would "centimeter" and "inches" also work? What happens when the user enters a wrong unit? Let's make another storyboard to demonstrate error handling.

### Handling Unknown Units (needs improvement)

What unit do you want to convert from? **cm**

What unit do you want to convert to? **inches**

Sorry, unknown unit.

What unit do you want to convert to? **inch**

Sorry, unknown unit.

What unit do you want to convert to? **grrr**

To eliminate frustration, it is better to list the units that the user can supply.

From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): **cm**

To unit: **in** No need to list the units again

We switched to a shorter prompt to make room for all the unit names. Exercise R4.24 explores a different alternative.

There is another issue that we haven't addressed yet. How does the user quit the program? The first storyboard suggests that the program will go on forever.

We can ask the user after seeing the sentinel that terminates an input sequence.

**Exiting the Program**

```
From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
Enter values, terminated by zero
30
30 cm = 11.81 in
0
More conversions (y, n)? n
(Program exits)
```

Sentinel triggers the prompt to exit

As you can see from this case study, a storyboard is essential for developing a working program. You need to know the flow of the user interaction in order to structure your program.

**SELF CHECK**

26. Provide a storyboard panel for a program that reads a number of test scores and prints the average score. The program only needs to process one set of scores. Don't worry about error handling.
27. Google has a simple interface for converting units. You just type the question, and you get the answer.

Google | How many inches in 30 cm | Search | Advanced Search

Web Show options... Results 1 - 10 of about 4,180,000 for How many Inches in 30 cm. (0.24 seconds)

30 centimeters = 11.8110236 inches  
More about calculator.

Make storyboards for an equivalent interface in a Java program. Show a scenario in which all goes well, and show the handling of two kinds of errors.

28. Consider a modification of the program in Self Check 26. Suppose we want to drop the lowest score before computing the average. Provide a storyboard for the situation in which a user only provides one score.
29. What is the problem with implementing the following storyboard in Java?

**Computing Multiple Averages**

```
Enter scores: 90 80 90 100 80
The average is 88
Enter scores: 100 70 70 100 80
The average is 84
Enter scores: -1
(Program exits)
```

-1 is used as a sentinel to exit the program

30. Produce a storyboard for a program that compares the growth of a \$10,000 investment for a given number of years under two interest rates.

**Practice It** Now you can try these exercises at the end of the chapter: R4.24, R4.25, R4.26.

## 4.7 Common Loop Algorithms

In the following sections, we discuss some of the most common algorithms that are implemented as loops. You can use them as starting points for your loop designs.

### 4.7.1 Sum and Average Value

To compute an average, keep a total and a count of all values.

Computing the sum of a number of inputs is a very common task. Keep a *running total*, a variable to which you add each input value. Of course, the total should be initialized with 0.

```
double total = 0;
while (in.hasNextDouble())
{
 double input = in.nextDouble();
 total = total + input;
}
```

Note that the `total` variable is declared outside the loop. We want the loop to update a single variable. The `input` variable is declared inside the loop. A separate variable is created for each input and removed at the end of each loop iteration.

To compute an average, count how many values you have, and divide by the count. Be sure to check that the count is not zero.

```
double total = 0;
int count = 0;
while (in.hasNextDouble())
{
 double input = in.nextDouble();
 total = total + input;
 count++;
}
double average = 0;
if (count > 0)
{
 average = total / count;
}
```

### 4.7.2 Counting Matches

To count values that fulfill a condition, check all values and increment a counter for each match.

You often want to know how many values fulfill a particular condition. For example, you may want to count how many spaces are in a string. Keep a *counter*, a variable that is initialized with 0 and incremented whenever there is a match.

```
int spaces = 0;
for (int i = 0; i < str.length(); i++)
{
 char ch = str.charAt(i);
 if (ch == ' ')
 {
 spaces++;
 }
}
```

For example, if `str` is "My Fair Lady", `spaces` is incremented twice (when `i` is 2 and 7).

Note that the `spaces` variable is declared outside the loop. We want the loop to update a single variable. The `ch` variable is declared inside the loop. A separate variable is created for each iteration and removed at the end of each loop iteration.

This loop can also be used for scanning inputs. The following loop reads text, a word at a time, and counts the number of words with at most three letters:

```
int shortWords = 0;
while (in.hasNext())
{
 String input = in.next();
 if (input.length() <= 3)
 {
 shortWords++;
 }
}
```

*In a loop that counts matches, a counter is incremented whenever a match is found.*



© Hiob/iStockphoto.

### 4.7.3 Finding the First Match

If your goal is to find a match, exit the loop when the match is found.

When you count the values that fulfill a condition, you need to look at all values. However, if your task is to find a match, then you can stop as soon as the condition is fulfilled.

Here is a loop that finds the first letter A in a string. Because we do not visit all elements in the string, a `while` loop is a better choice than a `for` loop:

```
boolean found = false;
char ch = '?';
int position = 0;
while (!found && position < str.length())
{
 ch = str.charAt(position);
 if (ch == 'A' || ch == 'a') { found = true; }
 else { position++; }
}
```

If a match was found, then `found` is `true`, `ch` is the first matching character, and `position` is the index of the first match. If the loop did not find a match, then `found` remains `false` after the end of the loop.

Note that the variable `ch` is declared *outside* the `while` loop because you may want to use the input after the loop has finished. If it had been declared inside the loop body, you would not be able to use it outside the loop.



© arflet/iStockphoto.

*When searching, you look at items until a match is found.*

#### 4.7.4 Prompting Until a Match is Found

In the preceding example, we searched a string for a character that matches a condition. You can apply the same process to user input. Suppose you are asking a user to enter a positive value  $< 100$ . Keep asking until the user provides a correct input:

```
boolean valid = false;
double input = 0;
while (!valid)
{
 System.out.print("Please enter a positive value < 100: ");
 input = in.nextDouble();
 if (0 < input && input < 100) { valid = true; }
 else { System.out.println("Invalid input."); }
}
```

Note that the variable `input` is declared *outside* the `while` loop because you will want to use the `input` after the loop has finished.

#### 4.7.5 Maximum and Minimum

To find the largest value, update the largest value seen so far whenever you see a larger one.

To compute the largest value in a sequence, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one.

```
double largest = in.nextDouble();
while (in.hasNextDouble())
{
 double input = in.nextDouble();
 if (input > largest)
 {
 largest = input;
 }
}
```

This algorithm requires that there is at least one input.

To compute the smallest value, simply reverse the comparison:

```
double smallest = in.nextDouble();
while (in.hasNextDouble())
{
 double input = in.nextDouble();
 if (input < smallest)
 {
 smallest = input;
 }
}
```

*To find the height of the tallest bus rider, remember the largest value so far, and update it whenever you see a taller one.*



### 4.7.6 Comparing Adjacent Values

To compare adjacent inputs, store the preceding input in a variable.

When processing a sequence of values in a loop, you sometimes need to compare a value with the value that just preceded it. For example, suppose you want to check whether a sequence of inputs contains adjacent duplicates such as 1 7 2 9 9 4 9.

Now you face a challenge. Consider the typical loop for reading a value:

```
double input;
while (in.hasNextDouble())
{
 input = in.nextDouble();
 ...
}
```

How can you compare the current input with the preceding one? At any time, `input` contains the current input, overwriting the previous one.

The answer is to store the previous input:

```
double input = 0;
while (in.hasNextDouble())
{
 double previous = input;
 input = in.nextDouble();
 if (input == previous)
 {
 System.out.println("Duplicate input");
 }
}
```

One problem remains. When the loop is entered for the first time, `input` has not yet been read. You can solve this problem with an initial input operation outside the loop:

```
double input = in.nextDouble();
while (in.hasNextDouble())
{
 double previous = input;
 input = in.nextDouble();
 if (input == previous)
 {
 System.out.println("Duplicate input");
 }
}
```



© tingberg/iStockphoto

*When comparing adjacent values, store the previous value in a variable.*

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program that demonstrates common loop algorithms.



#### SELF CHECK

31. What total is computed when no user input is provided in the algorithm in Section 4.7.1?
32. How do you compute the total of all positive inputs?
33. What are the values of `position` and `ch` when no match is found in the algorithm in Section 4.7.3?
34. What is wrong with the following loop for finding the position of the first space in a string?

```
boolean found = false;
for (int position = 0; !found && position < str.length(); position++)
{
 char ch = str.charAt(position);
 if (ch == ' ') { found = true; }
}
```

- 35.** How do you find the position of the *last* space in a string?
- 36.** What happens with the algorithm in Section 4.7.6 when no input is provided at all? How can you overcome that problem?

**Practice It** Now you can try these exercises at the end of the chapter: E4.5, E4.9, E4.10.

## HOW TO 4.1



### Writing a Loop

This How To walks you through the process of implementing a loop statement. We will illustrate the steps with the following example problem.

**Problem Statement** Read twelve temperature values (one for each month), and display the number of the month with the highest temperature. For example, according to <http://worldclimate.com>, the average maximum temperatures for Death Valley are (in order by month, in degrees Celsius):

18.2 22.6 26.4 31.1 36.6 42.2 45.7 44.5 40.2 33.1 24.2 17.6

In this case, the month with the highest temperature (45.7 degrees Celsius) is July, and the program should display 7.



© Stevegeer/iStockphoto.

#### Step 1 Decide what work must be done *inside* the loop.

Every loop needs to do some kind of repetitive work, such as

- Reading another item.
- Updating a value (such as a bank balance or total).
- Incrementing a counter.

If you can't figure out what needs to go inside the loop, start by writing down the steps that you would take if you solved the problem by hand. For example, with the temperature reading problem, you might write

**Read the first value.**

**Read the second value.**

**If the second value is higher than the first value**

**Set highest temperature to the second value.**

**Set highest month to 2.**

**Read the next value.**

**If the value is higher than the first and second values**

**Set highest temperature to the value.**

**Set highest month to 3.**

**Read the next value.**

**If the value is higher than the highest temperature seen so far**

**Set highest temperature to the value.**

**Set highest month to 4.**

...

Now look at these steps and reduce them to a set of *uniform* actions that can be placed into the loop body. The first action is easy:

**Read the next value.**

The next action is trickier. In our description, we used tests “higher than the first”, “higher than the first and second”, “higher than the highest temperature seen so far”. We need to settle on one test that works for all iterations. The last formulation is the most general.

Similarly, we must find a general way of setting the highest month. We need a variable that stores the current month, running from 1 to 12. Then we can formulate the second loop action:

```
If the value is higher than the highest temperature
 Set highest temperature to the value.
 Set highest month to current month.
```

Altogether our loop is

```
While ...
 Read the next value.
 If the value is higher than the highest temperature,
 set the highest temperature to the value.
 Set highest month to current month.
 Increment current month.
```

### Step 2 Specify the loop condition.

What goal do you want to reach in your loop? Typical examples are

- Has a counter reached its final value?
- Have you read the last input value?
- Has a value reached a given threshold?

In our example, we simply want the current month to reach 12.

### Step 3 Determine the loop type.

We distinguish between two major loop types. A *count-controlled* loop is executed a definite number of times. In an *event-controlled* loop, the number of iterations is not known in advance—the loop is executed until some event happens.

Count-controlled loops can be implemented as *for* statements. For other loops, consider the loop condition. Do you need to complete one iteration of the loop body before you can tell when to terminate the loop? In that case, choose a *do* loop. Otherwise, use a *while* loop.

Sometimes, the condition for terminating a loop changes in the middle of the loop body. In that case, you can use a Boolean variable that specifies when you are ready to leave the loop. Such a variable is called a *flag*. Follow this pattern:

```
boolean done = false;
while (!done)
{
 Do some work.
 If all work has been completed
 {
 done = true;
 }
 else
 {
 Do more work.
 }
}
```

In summary,

- If you know in advance how many times a loop is repeated, use a *for* loop.
- If the loop body must be executed at least once, use a *do* loop.
- Otherwise, use a *while* loop.

In our example, we read 12 temperature values. Therefore, we choose a *for* loop.

### Step 4 Set up variables for entering the loop for the first time.

List all variables that are used and updated in the loop, and determine how to initialize them. Commonly, counters are initialized with 0 or 1, totals with 0.

In our example, the variables are

**current month**  
**highest value**  
**highest month**

We need to be careful how we set up the highest temperature value. We can't simply set it to 0. After all, our program needs to work with temperature values from Antarctica, all of which may be negative.

A good option is to set the highest temperature value to the first input value. Of course, then we need to remember to read in only 11 more values, with the current month starting at 2.

We also need to initialize the highest month with 1. After all, in an Australian city, we may never find a month that is warmer than January.

### Step 5

Process the result after the loop has finished.

In many cases, the desired result is simply a variable that was updated in the loop body. For example, in our temperature program, the result is the highest month. Sometimes, the loop computes values that contribute to the final result. For example, suppose you are asked to average the temperatures. Then the loop should compute the sum, not the average. After the loop has completed, you are ready to compute the average: divide the sum by the number of inputs.

Here is our complete loop.

```
Read value.

highest temperature = value

highest month = 1

For current month from 2 to 12

 Read next value.

 If the value is higher than the highest temperature

 Set highest temperature to the value.

 Set highest month to current month.
```

### Step 6

Trace the loop with typical examples.

Hand trace your loop code, as described in Section 4.2. Choose example values that are not too complex—executing the loop 3–5 times is enough to check for the most common errors. Pay special attention when entering the loop for the first and last time.

Sometimes, you want to make a slight modification to make tracing feasible. For example, when hand-tracing the investment doubling problem, use an interest rate of 20 percent rather than 5 percent. When hand-tracing the temperature loop, use 4 data values, not 12.

Let's say the data are 22.6 36.6 44.5 24.2. Here is the walkthrough:

| current month | current value | highest month | highest temperature |
|---------------|---------------|---------------|---------------------|
|               |               | X             | 22.6                |
| 2             | 36.6          | 2             | 36.6                |
| 3             | 44.5          | 3             | 44.5                |
| 4             | 24.2          |               |                     |
|               |               |               |                     |
|               |               |               |                     |
|               |               |               |                     |

The trace demonstrates that **highest month** and **highest temperature** are properly set.

### Step 7

Implement the loop in Java.

Here's the loop for our example. Exercise E4.4 asks you to complete the program.

```
double highestTemperature;

highestTemperature = in.nextDouble();

int highestMonth = 1;
```

```

for (int currentMonth = 2; currentMonth <= 12; currentMonth++)
{
 double nextValue = in.nextDouble();
 if (nextValue > highestTemperature)
 {
 highestTemperature = nextValue;
 highestMonth = currentMonth;
 }
}
System.out.println(highestMonth);

```

**WORKED EXAMPLE 4.1****Credit Card Processing**

Learn how to use a loop to remove spaces from a credit card number. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 4.1.



© MorePixels/  
iStockphoto.

## 4.8 Nested Loops

When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

In Section 3.4, you saw how to nest two `if` statements. Similarly, complex iterations sometimes require a **nested loop**: a loop inside another loop statement. When processing tables, nested loops occur naturally. An outer loop iterates over all rows of the table. An inner loop deals with the columns in the current row.

In this section you will see how to print a table. For simplicity, we will simply print the powers of  $x$ ,  $x^n$ , as in the table at right.

Here is the pseudocode for printing the table:

```

Print table header.
For x from 1 to 10
 Print table row.
 Print new line.

```

How do you print a table row? You need to print a value for each exponent. This requires a second loop.

```

For n from 1 to 4
 Print xn.

```

This loop must be placed inside the preceding loop. We say that the inner loop is *nested* inside the outer loop.

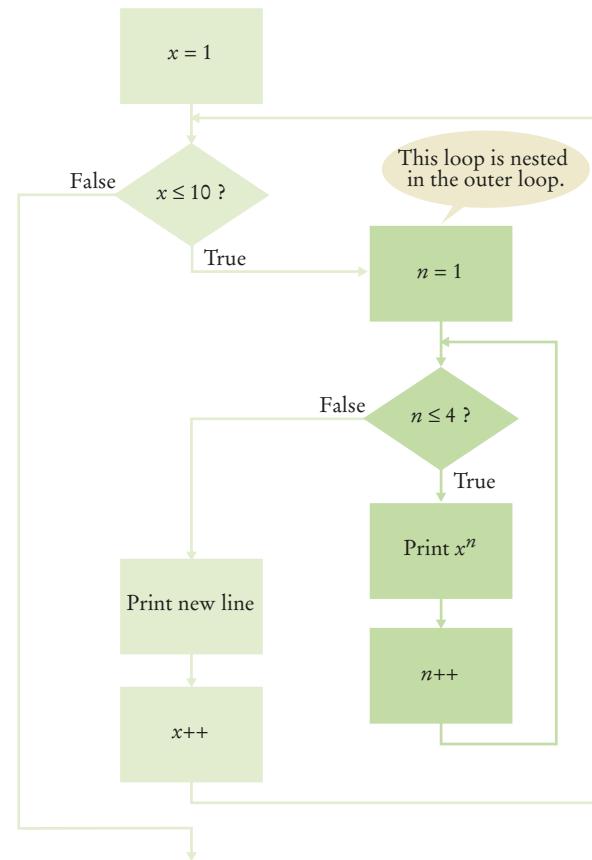
| $x^1$ | $x^2$ | $x^3$ | $x^4$ |
|-------|-------|-------|-------|
| 1     | 1     | 1     | 1     |
| 2     | 4     | 8     | 16    |
| 3     | 9     | 27    | 81    |
| ...   | ...   | ...   | ...   |
| 10    | 100   | 1000  | 10000 |

The hour and minute displays in a digital clock are an example of nested loops. The hours loop 12 times, and for each hour, the minutes loop 60 times.



© davejkahn/iStockphoto.

**Figure 7**  
Flowchart of a Nested Loop



There are 10 rows in the outer loop. For each  $x$ , the program prints four columns in the inner loop (see Figure 7). Thus, a total of  $10 \times 4 = 40$  values are printed.

Following is the complete program. Note that we also use loops to print the table header. However, those loops are not nested.

### sec08/PowerTable.java

```

1 /**
2 * This program prints a table of powers of x.
3 */
4 public class PowerTable
5 {
6 public static void main(String[] args)
7 {
8 final int NMAX = 4;
9 final double XMAX = 10;
10
11 // Print table header
12
13 for (int n = 1; n <= NMAX; n++)
14 {
15 System.out.printf("%10d", n);
16 }
17 System.out.println();

```

```

18 for (int n = 1; n <= NMAX; n++)
19 {
20 System.out.printf("%10s", "x ");
21 }
22 System.out.println();
23
24 // Print table body
25
26 for (double x = 1; x <= XMAX; x++)
27 {
28 // Print table row
29
30 for (int n = 1; n <= NMAX; n++)
31 {
32 System.out.printf("%10.0f", Math.pow(x, n));
33 }
34 System.out.println();
35 }
36 }
37 }
```

**Program Run**

| 1<br>x | 2<br>x | 3<br>x | 4<br>x |
|--------|--------|--------|--------|
| 1      | 1      | 1      | 1      |
| 2      | 4      | 8      | 16     |
| 3      | 9      | 27     | 81     |
| 4      | 16     | 64     | 256    |
| 5      | 25     | 125    | 625    |
| 6      | 36     | 216    | 1296   |
| 7      | 49     | 343    | 2401   |
| 8      | 64     | 512    | 4096   |
| 9      | 81     | 729    | 6561   |
| 10     | 100    | 1000   | 10000  |

**SELF CHECK**

37. Why is there a statement `System.out.println();` in the outer loop but not in the inner loop of the `PowerTable.java` program?
38. How would you change the program to display all powers from  $x^0$  to  $x^5$ ?
39. If you make the change in Self Check 38, how many values are displayed?
40. What do the following nested loops display?

```

for (int i = 0; i < 3; i++)
{
 for (int j = 0; j < 4; j++)
 {
 System.out.print(i + j);
 }
 System.out.println();
}
```

41. Write nested loops that make the following pattern of brackets:

```

[] [] [] []
[] [] [] []
[] [] [] []
```

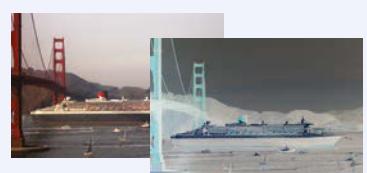
**Practice It** Now you can try these exercises at the end of the chapter: R4.30, E4.18, E4.20.

**Table 3** Nested Loop Examples

| Nested Loops                                                                                                                                                                          | Output                   | Explanation                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------|----------------------------------------------------------|
| <pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 4; j++) { Print "*" }     System.out.println(); }</pre>                                                                    | ****<br>****<br>****     | Prints 3 rows of 4 asterisks each.                       |
| <pre>for (i = 1; i &lt;= 4; i++) {     for (j = 1; j &lt;= 3; j++) { Print "*" }     System.out.println(); }</pre>                                                                    | ***<br>***<br>***<br>*** | Prints 4 rows of 3 asterisks each.                       |
| <pre>for (i = 1; i &lt;= 4; i++) {     for (j = 1; j &lt;= i; j++) { Print "*" }     System.out.println(); }</pre>                                                                    | *                        | Prints 4 rows of lengths 1, 2, 3, and 4.                 |
| <pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 5; j++)     {         if (j % 2 == 0) { Print "*" }         else { Print "-" }     }     System.out.println(); }</pre>     | -*-<br>-*-<br>-*-        | Prints asterisks in even columns, dashes in odd columns. |
| <pre>for (i = 1; i &lt;= 3; i++) {     for (j = 1; j &lt;= 5; j++)     {         if (i % 2 == j % 2) { Print "*" }         else { Print " " }     }     System.out.println(); }</pre> | * * *<br>* *<br>* * *    | Prints a checkerboard pattern.                           |

**WORKED EXAMPLE 4.2****Manipulating the Pixels in an Image**

This Worked Example shows how to use nested loops for manipulating the pixels in an image. The outer loop traverses the rows of the image, and the inner loop accesses each pixel of a row. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 4.2.



Cay Horstmann.

## 4.9 Problem Solving: Solve a Simpler Problem First

When developing a solution to a complex problem, first solve a simpler task.

As you learn more about programming, the complexity of the tasks that you are asked to solve will increase. When you face a complex task, you should apply an important skill: simplifying the problem, and solving the simpler problem first.

This is a good strategy for several reasons. Usually, you learn something useful from solving the simpler task. Moreover, the complex problem can seem unsurmountable, and you may find it difficult to know where to get started. When you are successful with a simpler problem first, you will be much more motivated to try the harder one.

It takes practice and a certain amount of courage to break down a problem into a sequence of simpler ones. The best way to learn this strategy is to practice it. When you work on your next assignment, ask yourself what is the absolutely simplest part of the task that is helpful for the end result, and start from there. With some experience, you will be able to design a plan that builds up a complete solution as a manageable sequence of intermediate steps.

Let us look at an example. You are asked to arrange pictures, lining them up along the top edges, separating them with small gaps, and starting a new row whenever you run out of room in the current row.



National Gallery of Art. (See page C1.)

We use the `Picture` class of Worked Example 4.2. It has methods

```
public void load(String source)
public int getWidth()
public int getHeight()
public void move(int dx, int dy)
```

Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

Instead of tackling the entire assignment at once, here is a plan that solves a series of simpler problems.

1. Draw one picture.



2. Draw two pictures next to each other.



3. Draw two pictures with a gap between them.



4. Draw all pictures in a long row.



5. Draw a row of pictures until you run out of room, then put one more picture in the next row.



Let's get started with this plan.

1. The purpose of the first step is to become familiar with the Picture class. As it turns out, the pictures are in files picture1.jpg ... picture20.jpg. Let's load the first one:

```
public class Gallery1
{
 public static void main(String[] args)
 {
 Picture pic = new Picture();
 pic.load("picture1.jpg");
 }
}
```

That's enough to show the picture.



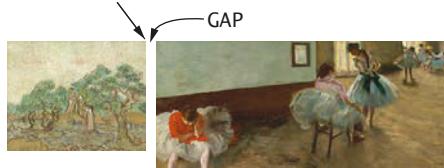
2. Now let's put the next picture after the first. We need to move it to the right-most *x*-coordinate of the preceding picture.



```
Picture pic = new Picture();
pic.load("picture1.jpg");
Picture pic2 = new Picture();
pic2.load("picture2.jpg");
pic2.move(pic.getWidth(), 0);
```

3. The next step is to separate the two by a small gap when the second is moved:

```
pic.getWidth()
```



```
final int GAP = 10;
```

```
Picture pic = new Picture();
pic.load("picture1.jpg");
Picture pic2 = new Picture();
pic2.load("picture2.jpg");
int x = pic.getWidth() + GAP;
pic2.move(x, 0);
```

4. Now let's put all pictures in a row. Read the pictures in a loop, and then put each picture to the right of the one that preceded it. In the loop, you need to track two pictures: the one that is being read in, and the one that preceded it (see Section 4.7.6).



```
final int GAP = 10;
final int PICTURES = 20;

Picture pic = new Picture();
pic.load("picture1.jpg");
int x = 0;
for (int i = 2; i <= PICTURES; i++)
{
 Picture previous = pic;
 pic = new Picture();
 pic.load("picture" + i + ".jpg");
 x = x + previous.getWidth() + GAP;
 pic.move(x, 0);
}
```

5. Of course, we don't want to have all pictures in a row. The right margin of a picture should not extend past MAX\_WIDTH.

```
x = x + previous.getWidth() + GAP;
if (x + pic.getWidth() < MAX_WIDTH)
{
 Place pic on current row.
}
else
{
 Place pic on next row.
}
```

If the image doesn't fit any more, then we need to put it on the next row, below all the pictures in the current row. We'll set a variable `maxY` to the maximum

*y*-coordinate of all placed pictures, updating it whenever a new picture is placed:

```
maxY = Math.max(maxY, pic.getHeight());
```



The following statement places a picture on the next row:

```
pic.move(0, maxY + GAP);
```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) for the listings of all preliminary stages of the gallery program.

Now we have written complete programs for all preliminary stages. We know how to line up the pictures, how to separate them with gaps, how to find out when to start a new row, and where to start it.

With this knowledge, producing the final version is straightforward. Here is the program listing.

#### sec09/Gallery6.java

```

1 public class Gallery6
2 {
3 public static void main(String[] args)
4 {
5 final int MAX_WIDTH = 720;
6 final int GAP = 10;
7 final int PICTURES = 20;
8
9 Picture pic = new Picture();
10 pic.load("picture1.jpg");
11 int x = 0;
12 int y = 0;
13 int maxY = 0;
14
15 for (int i = 2; i <= 20; i++)
16 {
17 maxY = Math.max(maxY, y + pic.getHeight());
18 Picture previous = pic;
19 pic = new Picture();
20 pic.load("picture" + i + ".jpg");
21 x = x + previous.getWidth() + GAP;
22 if (x + pic.getWidth() >= MAX_WIDTH)
23 {
24 x = 0;
25 y = maxY + GAP;
26 }
27 pic.move(x, y);
28 }
29 }
30 }
```



42. Suppose you are asked to find all words in which no letter is repeated from a list of words. What simpler problem could you try first?
43. You need to write a program for DNA analysis that checks whether a substring of one string is contained in another string. What simpler problem can you solve first?
44. You want to remove “red eyes” from images and are looking for red circles. What simpler problem can you start with?
45. Consider the task of finding numbers in a string. For example, the string “In 1987, a typical personal computer cost \$3,000 and had 512 kilobytes of RAM.” has three numbers. Break this task down into a sequence of simpler tasks.

**Practice It** Now you can try these exercises at the end of the chapter: P4.6, P4.7.

## 4.10 Application: Random Numbers and Simulations

In a simulation, you use the computer to simulate an activity.

A *simulation program* uses the computer to simulate an activity in the real world (or an imaginary one). Simulations are commonly used for predicting climate change, analyzing traffic, picking stocks, and many other applications in science and business. In many simulations, one or more loops are used to modify the state of a system and observe the changes. You will see examples in the following sections.

### 4.10.1 Generating Random Numbers

Many events in the real world are difficult to predict with absolute precision, yet we can sometimes know the average behavior quite well. For example, a store may know from experience that a customer arrives every five minutes. Of course, that is an average—customers don’t arrive in five minute intervals. To accurately model customer traffic, you want to take that random fluctuation into account. Now, how can you run such a simulation in the computer?

The Java library has a *random number generator*, which produces numbers that appear to be completely random. Calling `Math.random()` yields a random floating-point number that is  $\geq 0$  and  $< 1$ . Call `Math.random()` again, and you get a different number.

The following program calls `Math.random()` ten times.

#### sec10\_01/RandomDemo.java

```

1 /**
2 * This program prints ten random numbers between 0 and 1.
3 */
4 public class RandomDemo
5 {
6 public static void main(String[] args)
7 {
8 for (int i = 1; i <= 10; i++)
9 {
10 double r = Math.random();
11 System.out.println(r);
12 }
13 }
14 }
```

```

12 }
13 }
14 }
```

### Program Run

```

0.6513550469421886
0.920193662882893
0.6904776061289993
0.8862828776788884
0.7730177555323139
0.3020238718668635
0.0028504531690907164
0.9099983981705169
0.1151636530517488
0.1592258808929058
```

Actually, the numbers are not completely random. They are drawn from sequences of numbers that don't repeat for a long time. These sequences are actually computed from fairly simple formulas; they just behave like random numbers (see Exercise P4.11). For that reason, they are often called **pseudorandom** numbers.

## 4.10.2 Simulating Die Tosses

In actual applications, you need to transform the output from the random number generator into different ranges. For example, to simulate the throw of a die, you need random integers between 1 and 6.

Here is the general recipe for computing random integers between two bounds  $a$  and  $b$ . As you know from Programming Tip 4.3 on page 158, there are  $b - a + 1$  values between  $a$  and  $b$ , including the bounds themselves. First compute



© ktsimage/iStockphoto.

```
(int) (Math.random() * (b - a + 1))
```

to obtain a random integer between 0 and  $b - a$ , then add  $a$ , yielding a random value between  $a$  and  $b$ :

```
int r = (int) (Math.random() * (b - a + 1)) + a;
```

Here is a program that simulates the throw of a pair of dice:

### sec10\_02/Dice.java

```

1 /**
2 * This program simulates tosses of a pair of dice.
3 */
4 public class Dice
5 {
6 public static void main(String[] args)
7 {
8 for (int i = 1; i <= 10; i++)
9 {
10 // Generate two random numbers between 1 and 6
11 int d1 = (int) (Math.random() * 6) + 1;
12 int d2 = (int) (Math.random() * 6) + 1;
```

```

14 System.out.println(d1 + " " + d2);
15 }
16 System.out.println();
17 }
18 }
```

**Program Run**

```

5 1
2 1
1 2
5 1
1 2
6 4
4 4
6 1
6 3
5 2
```

**4.10.3 The Monte Carlo Method**

The Monte Carlo method is an ingenious method for finding approximate solutions to problems that cannot be precisely solved. (The method is named after the famous casino in Monte Carlo.) Here is a typical example. It is difficult to compute the number  $\pi$ , but you can approximate it quite well with the following simulation.

Simulate shooting a dart into a square surrounding a circle of radius 1. That is easy: generate random  $x$  and  $y$  coordinates between  $-1$  and  $1$ .

If the generated point lies inside the circle, we count it as a *hit*. That is the case when  $x^2 + y^2 \leq 1$ . Because our shots are entirely random, we expect that the ratio of *hits / tries* is approximately equal to the ratio of the areas of the circle and the square, that is,  $\pi / 4$ . Therefore, our estimate for  $\pi$  is  $4 \times \text{hits} / \text{tries}$ . This method yields an estimate for  $\pi$ , using nothing but simple arithmetic.

To generate a random floating-point value between  $-1$  and  $1$ , you compute:

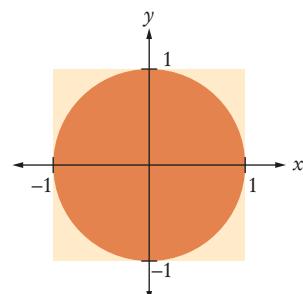
```

double r = Math.random(); // 0 ≤ r < 1
double x = -1 + 2 * r; // -1 ≤ x < 1
```

As  $r$  ranges from  $0$  (inclusive) to  $1$  (exclusive),  $x$  ranges from  $-1 + 2 \times 0 = -1$  (inclusive) to  $-1 + 2 \times 1 = 1$  (exclusive). In our application, it does not matter that  $x$  never reaches  $1$ . The points that fulfill the equation  $x = 1$  lie on a line with area  $0$ .



© timstarkkey/iStockphoto.



Here is the program that carries out the simulation:

### sec10\_03/MonteCarlo.java

```

1 /**
2 * This program computes an estimate of pi by simulating dart throws onto a square.
3 */
4 public class MonteCarlo
5 {
6 public static void main(String[] args)
7 {
8 final int TRIES = 10000;
9
10 int hits = 0;
11 for (int i = 1; i <= TRIES; i++)
12 {
13 // Generate two random numbers between -1 and 1
14
15 double r = Math.random();
16 double x = -1 + 2 * r; // Between -1 and 1
17 r = Math.random();
18 double y = -1 + 2 * r;
19
20 // Check whether the point lies in the unit circle
21
22 if (x * x + y * y <= 1) { hits++; }
23 }
24
25 /*
26 The ratio hits / tries is approximately the same as the ratio
27 circle area / square area = pi / 4
28 */
29
30 double piEstimate = 4.0 * hits / TRIES;
31 System.out.println("Estimate for pi: " + piEstimate);
32 }
33 }
```

### Program Run

Estimate for pi: 3.1504



46. How do you simulate a coin toss with the `Math.random()` method?
47. How do you simulate the picking of a random playing card?
48. Why does the loop body in `Dice.java` call `Math.random()` twice?
49. In many games, you throw a pair of dice to get a value between 2 and 12. What is wrong with this simulated throw of a pair of dice?  
`int sum = (int) (Math.random() * 11) + 2;`
50. How do you generate a random floating-point number  $\geq 0$  and  $< 100$ ?

**Practice It** Now you can try these exercises at the end of the chapter: R4.31, E4.7, P4.10.

## Special Topic 4.3

**Drawing Graphical Shapes**

In Java, it is easy to produce simple drawings such as the one in Figure 8. By writing programs that draw such patterns, you can practice programming loops. For now, we give you a program outline into which you place your drawing code. The program outline also contains the necessary code for displaying a window containing your drawing. You need not look at that code now. It will be discussed in detail in Chapter 10.

Your drawing instructions go inside the `draw` method:

```
public class TwoRowsOfSquares
{
 public static void draw(Graphics g)
 {
 Draw the desired shapes.
 }
 . . .
}
```

When the window is shown, the `draw` method is called, and your drawing instructions will be executed.



**Figure 8** Two Rows of Squares

**Table 4** Graphics Methods

| Method                                       | Result                                                                              | Notes                                                                                                                                                   |
|----------------------------------------------|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>g.drawRect(x, y, width, height)</code> |   | (x, y) is the top left corner.                                                                                                                          |
| <code>g.drawOval(x, y, width, height)</code> |  | (x, y) is the top left corner of the box that bounds the ellipse. To draw a circle, use the same value for width and height.                            |
| <code>g.fillRect(x, y, width, height)</code> |  | The rectangle is filled in.                                                                                                                             |
| <code>g.fillOval(x, y, width, height)</code> |  | The oval is filled in.                                                                                                                                  |
| <code>g.drawLine(x1, y1, x2, y2)</code>      |  | (x1, y1) and (x2, y2) are the endpoints.                                                                                                                |
| <code>g.drawString("Message", x, y)</code>   |  | (x, y) is the basepoint.                                                                                                                                |
| <code>g.setColor(color)</code>               | From now on, draw or fill methods will use this color.                              | Use <code>Color.RED</code> , <code>Color.GREEN</code> , <code>Color.BLUE</code> , and so on. (See Table 10.1 for a complete list of predefined colors.) |

The draw method receives an object of type `Graphics`. The `Graphics` object has methods for drawing shapes. It also remembers the color that is used for drawing operations. You can think of the `Graphics` object as the equivalent of `System.out` for drawing shapes instead of printing values. Table 4 shows useful methods of the `Graphics` class.

The program below draws the squares shown in Figure 8. When you want to produce your own drawings, make a copy of this program and modify it. Replace the drawing tasks in the `draw` method. Rename the class (for example, `Spiral` instead of `TwoRowsOfSquares`).

### **special\_topic\_3/TwoRowsOfSquares.java**

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3 import javax.swing.JFrame;
4 import javax.swing.JComponent;
5
6 /**
7 * This program draws two rows of squares.
8 */
9 public class TwoRowsOfSquares
10 {
11 public static void draw(Graphics g)
12 {
13 final int width = 20;
14 g.setColor(Color.BLUE);
15
16 // Top row. Note that the top left corner of the drawing has coordinates (0, 0)
17 int x = 0;
18 int y = 0;
19 for (int i = 0; i < 10; i++)
20 {
21 g.fillRect(x, y, width, width);
22 x = x + 2 * width;
23 }
24 // Second row, offset from the first one
25 x = width;
26 y = width;
27 for (int i = 0; i < 10; i++)
28 {
29 g.fillRect(x, y, width, width);
30 x = x + 2 * width;
31 }
32 }
33
34 public static void main(String[] args)
35 {
36 // Do not look at the code in the main method
37 // Your code will go into the draw method above
38
39 JFrame frame = new JFrame();
40
41 final int FRAME_WIDTH = 400;
42 final int FRAME_HEIGHT = 400;
43
44 frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
45 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46
47 JComponent component = new JComponent()
48 {
49 public void paintComponent(Graphics graph)
50 {

```

```

51 draw(graph);
52 }
53 };
54 }
55 frame.addComponent();
56 frame.setVisible(true);
57 }
58 }
```



## VIDEO EXAMPLE 4.2

**Drawing a Spiral**

In this Video Example, you will see how to develop a program that draws a spiral. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 4.2.



© Rpsycho/iStockphoto.

***Computing & Society 4.2*** Digital Piracy

As you read this, you will have written a few computer programs and experienced firsthand how much effort it takes to write even the humblest of programs. Writing a real software product, such as a financial application or a computer game, takes a lot of time and money. Few people, and fewer companies, are going to spend that kind of time and money if they don't have a reasonable chance to make more money from their effort. (Actually, some companies give away their software in the hope that users will click on advertisements or upgrade to more elaborate paid versions. Other companies give away the software that enables users to read and use files but sell the software needed to create those files. Finally, there are individuals who donate their time, out of enthusiasm, and produce programs that you can copy freely.)

When selling software, a company must rely on the honesty of its customers. It is an easy matter for an unscrupulous person to make copies of computer programs without paying for them. In most countries that is illegal. Most governments provide legal protection, such as copyright laws and patents, to encourage the development of new products. Countries that tolerate widespread piracy have found that they have an ample cheap supply

of foreign software, but no local manufacturers willing to design good software for their own citizens, such as word processors in the local script or financial programs adapted to the local tax laws.

When a mass market for software first appeared, vendors were enraged by the money they lost through piracy. They tried to fight back to ensure that only the legitimate owner could use the software by using various schemes, such as *dongles*—devices that must be attached to a printer port before the software will run. Legitimate users hated these measures. They paid for the software, but they had to suffer through inconveniences, such as having multiple dongles sticking out from their computer.

Because it is so easy and inexpensive to pirate software, and the chance of being found out is minimal, you have to make a moral choice for yourself. If a package that you would really like to have is too expensive for your budget, do you steal it, or do you stay honest and get by with a more affordable product?

Of course, piracy is not limited to software. The same issues arise for other digital products as well. You may have had the opportunity to obtain copies of songs or movies without payment. Or you may have been frustrated by a copy protection device

on your music player that made it difficult for you to listen to songs that you paid for. Admittedly, it can be difficult to have a lot of sympathy for a musical ensemble whose publisher charges a lot of money for what seems to have been very little effort on their part, at least when compared to the effort that goes into designing and implementing a software package. Nevertheless, it seems only fair that artists and authors receive some compensation for their efforts.

How to pay artists, authors, and programmers fairly, without burdening honest customers, is an unsolved problem at the time of this writing, and many computer scientists are engaged in research in this area.



© RapidEye/iStockphoto.

## CHAPTER SUMMARY

### Explain the flow of execution in a loop.

- A loop executes instructions repeatedly while a condition is true.
- An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.



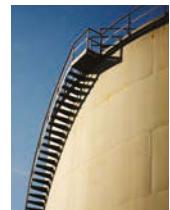
### Use the technique of hand-tracing to analyze the behavior of a program.

- Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.
- Hand-tracing can help you understand how an unfamiliar algorithm works.
- Hand-tracing can show errors in code or pseudocode.



### Use for loops for implementing count-controlled loops.

- The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.



### Choose between the while loop and the do loop.

- The do loop is appropriate when the loop body must be executed at least once.

### Implement loops that read sequences of input data.

- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a Boolean variable to control a loop. Set the variable to true before entering the loop, then set it to false to leave the loop.
- Use input redirection to read input from a file. Use output redirection to capture program output in a file.



### Use the technique of storyboarding for planning user interactions.

- A storyboard consists of annotated sketches for each step in an action sequence.
- Developing a storyboard helps you understand the inputs and outputs that are required for a program.

### Know the most common loop algorithms.

- To compute an average, keep a total and a count of all values.
- To count values that fulfill a condition, check all values and increment a counter for each match.

- If your goal is to find a match, exit the loop when the match is found.
- To find the largest value, update the largest value seen so far whenever you see a larger one.
- To compare adjacent inputs, store the preceding input in a variable.

### Use nested loops to implement multiple levels of iteration.



- When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.

### Design programs that carry out complex tasks.

- When developing a solution to a complex problem, first solve a simpler task.
- Make a plan consisting of a series of tasks, each a simple extension of the previous one, and ending with the original problem.

### Apply loops to the implementation of simulations.

- In a simulation, you use the computer to simulate an activity.
- You can introduce randomness by calling the random number generator.



## STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

java.awt.Color  
 java.awt.Graphics  
     drawLine  
     drawOval  
     drawRect  
     drawString  
     setColor

java.lang.Math  
     random

## REVIEW EXERCISES

### ■ R4.1 Given the variables

```
String stars = "*****";
String stripes = "=====";
```

what do these loops print?

- ```
int i = 0;
while (i < 5)
{
    System.out.println(stars.substring(0, i));
    i++;
}
```
- ```
int i = 0;
while (i < 5)
{
 System.out.print(stars.substring(0, i));
```

```

 System.out.println(stripes.substring(i, 5));
 i++;
 }
c. int i = 0;
 while (i < 10)
 {
 if (i % 2 == 0) { System.out.println(stars); }
 else { System.out.println(stripes); }
 }
}

```

**■ R4.2** What do these loops print?

- a.** int i = 0; int j = 10;  
while (i < j) { System.out.println(i + " " + j); i++; j--; }
- b.** int i = 0; int j = 10;  
while (i < j) { System.out.println(i + j); i++; j++; }

**■ R4.3** What do these code snippets print?

- a.** int result = 0;  
for (int i = 1; i <= 10; i++) { result = result + i; }  
System.out.println(result);
- b.** int result = 1;  
for (int i = 1; i <= 10; i++) { result = i - result; }  
System.out.println(result);
- c.** int result = 1;  
for (int i = 1; i > 0; i--) { result = result \* i; }  
System.out.println(result);
- d.** int result = 1;  
for (int i = 1; i <= 10; i = i \* 2) { result = result \* i; }  
System.out.println(result);

**■ R4.4** Write a while loop that prints

- a.** All squares less than n. For example, if n is 100, print 0 1 4 9 16 25 36 49 64 81.
- b.** All positive numbers that are divisible by 10 and less than n. For example, if n is 100, print 10 20 30 40 50 60 70 80 90
- c.** All powers of two less than n. For example, if n is 100, print 1 2 4 8 16 32 64.

**■■ R4.5** Write a loop that computes

- a.** The sum of all even numbers between 2 and 100 (inclusive).
- b.** The sum of all squares between 1 and 100 (inclusive).
- c.** The sum of all odd numbers between a and b (inclusive).
- d.** The sum of all odd digits of n. (For example, if n is 32677, the sum would be  $3 + 7 + 7 = 17$ .)

**■ R4.6** Provide trace tables for these loops.

- a.** int i = 0; int j = 10; int n = 0;  
while (i < j) { i++; j--; n++; }
- b.** int i = 0; int j = 0; int n = 0;  
while (i < 10) { i++; n = n + i + j; j++; }
- c.** int i = 10; int j = 0; int n = 0;  
while (i > 0) { i--; j++; n = n + i - j; }
- d.** int i = 0; int j = 10; int n = 0;  
while (i != j) { i = i + 2; j = j - 2; n++; }

**R4.7** What do these loops print?

- a. `for (int i = 1; i < 10; i++) { System.out.print(i + " "); }`
- b. `for (int i = 1; i < 10; i += 2) { System.out.print(i + " "); }`
- c. `for (int i = 10; i > 1; i--) { System.out.print(i + " "); }`
- d. `for (int i = 0; i < 10; i++) { System.out.print(i + " "); }`
- e. `for (int i = 1; i < 10; i = i * 2) { System.out.print(i + " "); }`
- f. `for (int i = 1; i < 10; i++) { if (i % 2 == 0) { System.out.print(i + " "); } }`

**R4.8** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?**R4.9** Write a program trace for the pseudocode in Exercise E4.6, assuming the input values are 4 7 –2 –5 0.**R4.10** What is an “off-by-one” error? Give an example from your own programming experience.**R4.11** What is a sentinel value? Give a simple rule when it is appropriate to use a numeric sentinel value.**R4.12** Which loop statements does Java support? Give simple rules for when to use each loop type.**R4.13** How many iterations do the following loops carry out? Assume that *i* is not changed in the loop body.

- a. `for (int i = 1; i <= 10; i++) . . .`
- b. `for (int i = 0; i < 10; i++) . . .`
- c. `for (int i = 10; i > 0; i--) . . .`
- d. `for (int i = -10; i <= 10; i++) . . .`
- e. `for (int i = 10; i >= 0; i++) . . .`
- f. `for (int i = -10; i <= 10; i = i + 2) . . .`
- g. `for (int i = -10; i <= 10; i = i + 3) . . .`

**R4.14** Write pseudocode for a program that prints a calendar such as the following:

| Su | M  | T  | W  | Th | F  | Sa |
|----|----|----|----|----|----|----|
|    |    |    | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 |    |

**R4.15** Write pseudocode for a program that prints a Celsius/Fahrenheit conversion table such as the following:

| Celsius |   | Fahrenheit |
|---------|---|------------|
| 0       |   | 32         |
| 10      |   | 50         |
| 20      |   | 68         |
| . . .   | . | . . .      |
| 100     |   | 212        |

- **R4.16** Write pseudocode for a program that reads a student record, consisting of the student's first and last name, followed by a sequence of test scores and a sentinel of  $-1$ . The program should print the student's average score. Then provide a trace table for this sample input:

```
Harry Morgan 94 71 86 95 -1
```

- **R4.17** Write pseudocode for a program that reads a sequence of student records and prints the total score for each student. Each record has the student's first and last name, followed by a sequence of test scores and a sentinel of  $-1$ . The sequence is terminated by the word `END`. Here is a sample sequence:

```
Harry Morgan 94 71 86 95 -1
Sally Lin 99 98 100 95 90 -1
END
```

Provide a trace table for this sample input.

- **R4.18** Rewrite the following `for` loop into a `while` loop.

```
int s = 0;
for (int i = 1; i <= 10; i++)
{
 s = s + i;
}
```

- **R4.19** Rewrite the following `do` loop into a `while` loop.

```
int n = in.nextInt();
double x = 0;
double s;
do
{
 s = 1.0 / (1 + n * n);
 n++;
 x = x + s;
}
while (s > 0.01);
```

- **R4.20** Provide trace tables of the following loops.

a. `int s = 1;`  
`int n = 1;`  
`while (s < 10) { s = s + n; }`  
`n++;`

b. `int s = 1;`  
`for (int n = 1; n < 5; n++)`  
`{`  
 `s = s + n;`  
`}`

c. `int s = 1;`  
`int n = 1;`  
`do`  
`{`  
 `s = s + n;`  
 `n++;`  
`}`  
`while (s < 10 * n);`

- **R4.21** What do the following loops print? Work out the answer by tracing the code, not by using the computer.

```

a. int s = 1;
 for (int n = 1; n <= 5; n++)
 {
 s = s + n;
 System.out.print(s + " ");
 }

b. int s = 1;
 for (int n = 1; s <= 10; System.out.print(s + " "))
 {
 n = n + 2;
 s = s + n;
 }

c. int s = 1;
 int n;
 for (n = 1; n <= 5; n++)
 {
 s = s + n;
 n++;
 }
 System.out.print(s + " " + n);

```

- **R4.22** What do the following program segments print? Find the answers by tracing the code, not by using the computer.

```

a. int n = 1;
 for (int i = 2; i < 5; i++) { n = n + i; }
 System.out.print(n);

b. int i;
 double n = 1 / 2;
 for (i = 2; i <= 5; i++) { n = n + 1.0 / i; }
 System.out.print(i);

c. double x = 1;
 double y = 1;
 int i = 0;
 do
 {
 y = y / 2;
 x = x + y;
 i++;
 }
 while (x < 1.8);
 System.out.print(i);

d. double x = 1;
 double y = 1;
 int i = 0;
 while (y >= 1.5)
 {
 x = x / 2;
 y = x + y;
 i++;
 }
 System.out.print(i);

```

- ■ **R4.23** Give an example of a for loop where symmetric bounds are more natural. Give an example of a for loop where asymmetric bounds are more natural.

- **R4.24** Add a storyboard panel for the conversion program in Section 4.6 on page 164 that shows a scenario where a user enters incompatible units.
- **R4.25** In Section 4.6, we decided to show users a list of all valid units in the prompt. If the program supports many more units, this approach is unworkable. Give a storyboard panel that illustrates an alternate approach: If the user enters an unknown unit, a list of all known units is shown.
- **R4.26** Change the storyboards in Section 4.6 to support a menu that asks users whether they want to convert units, see program help, or quit the program. The menu should be displayed at the beginning of the program, when a sequence of values has been converted, and when an error is displayed.
- **R4.27** Draw a flowchart for a program that carries out unit conversions as described in Section 4.6.
- **R4.28** In Section 4.7.5, the code for finding the largest and smallest input initializes the `largest` and `smallest` variables with an input value. Why can't you initialize them with zero?
- **R4.29** What are nested loops? Give an example where a nested loop is typically used.

■■ **R4.30** The nested loops

```
for (int i = 1; i <= height; i++)
{
 for (int j = 1; j <= width; j++) { System.out.print("*"); }
 System.out.println();
}
```

display a rectangle of a given width and height, such as

```



```

Write a *single* `for` loop that displays the same rectangle.

- **R4.31** Suppose you design an educational game to teach children how to read a clock. How do you generate random values for the hours and minutes?
- **R4.32** In a travel simulation, Harry will visit one of his friends that are located in three states. He has ten friends in California, three in Nevada, and two in Utah. How do you produce a random number between 1 and 3, denoting the destination state, with a probability that is proportional to the number of friends in each state?

## PRACTICE EXERCISES

- **E4.1** Write programs with loops that compute
  - a. The sum of all even numbers between 2 and 100 (inclusive).
  - b. The sum of all squares between 1 and 100 (inclusive).
  - c. All powers of 2 from  $2^0$  up to  $2^{20}$ .
  - d. The sum of all odd numbers between a and b (inclusive), where a and b are inputs.
  - e. The sum of all odd digits of an input. (For example, if the input is 32677, the sum would be  $3 + 7 + 7 = 17$ .)

- E4.2** Write programs that read a sequence of integer inputs and print
    - a. The smallest and largest of the inputs.
    - b. The number of even and odd inputs.
    - c. Cumulative totals. For example, if the input is 1 7 2 9, the program should print 1 8 10 19.
    - d. All adjacent duplicates. For example, if the input is 1 3 3 4 5 5 6 6 6 2, the program should print 3 5 6.
  
  - E4.3** Write programs that read a line of input as a string and print
    - a. Only the uppercase letters in the string.
    - b. Every second letter of the string.
    - c. The string, with all vowels replaced by an underscore.
    - d. The number of vowels in the string.
    - e. The positions of all vowels in the string.
  
  - E4.4** Complete the program in How To 4.1 on page 171. Your program should read twelve temperature values and print the month with the highest temperature.
  
  - E4.5** Write a program that reads a set of floating-point values. Ask the user to enter the values, then print
    - the average of the values.
    - the smallest of the values.
    - the largest of the values.
    - the range, that is the difference between the smallest and largest.
- Of course, you may only prompt for the values once.
- E4.6** Translate the following pseudocode for finding the minimum value from a set of inputs into a Java program.

```

Set a Boolean variable "first" to true.
If the scanner has more numbers
 Read the next value.
 If first is true
 Set the minimum to the value.
 Set first to false.
 Else if the value is less than the minimum
 Set the minimum to the value.
Print the minimum.

```

- E4.7** Translate the following pseudocode for randomly permuting the characters in a string into a Java program.

```

Read a word.
Repeat word.length() times
 Pick a random position i in the word, but not the last position.
 Pick a random position j > i in the word.
 Swap the letters at positions j and i.
Print the word.

```



To swap the letters, construct substrings as follows:



Then replace the string with

```
first + word.charAt(j) + middle + word.charAt(i) + last
```

- **E4.8** Write a program that reads a word and prints each character of the word on a separate line. For example, if the user provides the input "Harry", the program prints

```
H
a
r
r
y
```

- ■ **E4.9** Write a program that reads a word and prints the word in reverse. For example, if the user provides the input "Harry", the program prints

```
yrraH
```

- **E4.10** Write a program that reads a word and prints the number of vowels in the word. For this exercise, assume that a e i o u y are vowels. For example, if the user provides the input "Harry", the program prints 2 vowels.

- ■ ■ **E4.11** Write a program that reads a word and prints the number of syllables in the word. For this exercise, assume that syllables are determined as follows: Each sequence of adjacent vowels a e i o u y, except for the last e in a word, is a syllable. However, if that algorithm yields a count of 0, change it to 1. For example,

| Word  | Syllables |
|-------|-----------|
| Harry | 2         |
| hairy | 2         |
| hare  | 1         |
| the   | 1         |

- ■ ■ **E4.12** Write a program that reads a word and prints all substrings, sorted by length. For example, if the user provides the input "rum", the program prints

```
r
u
m
ru
um
rum
```

- ■ ■ **E4.13** Write a program that reads a string and prints the most frequently occurring letter. If there are multiple letters that occur with the same maximum frequency, print them all. For example, if the word is mississippi, print is because i and s occur four times each, and no letter occurs more frequently.

- ■ ■ **E4.14** Write a program that reads a string and prints the longest sequence of vowels. If there are multiple sequences of the same length, print them all. For example, if the word is oiseau, print eau, and if the word is teakwood, print ea and oo.

- E4.15** Write a program that reads a sequence of words and then prints them in a box, with each word centered, like this:

```
+-----+
| Hello |
| Java |
|programmer|
+-----+
```

- E4.16** Write a program that prints all powers of 2 from  $2^0$  up to  $2^{20}$ .

- E4.17** Write a program that reads a number and prints all of its *binary digits*: Print the remainder number % 2, then replace the number with number / 2. Keep going until the number is 0. For example, if the user provides the input 13, the output should be

```
1
0
1
1
```

- E4.18** Write a program that prints a multiplication table, like this:

|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 2  | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20  |
| 3  | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 | 30  |
| .  | .  | .  |    |    |    |    |    |    |     |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

- E4.19** Write a program that reads an integer and displays, using asterisks, a filled and hollow square, placed next to each other. For example if the side length is 5, the program should display

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

- E4.20** Write a program that reads an integer and displays, using asterisks, a filled diamond of the given side length. For example, if the side length is 4, the program should display

```

*
**

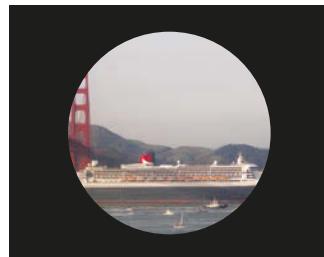
**
*
```

- E4.21** Using the Picture class from Worked Example 4.2, write a program that reads in the file names of two pictures and superimposes them. The result is a picture whose width and height are the larger of the widths and heights of both pictures. In the area where both pictures have pixels, average the colors.

- E4.22** Using the Picture class from Worked Example 4.2, write a program that reads in the file names of two pictures and superimposes them. The result is a picture whose width and height are the larger of the widths and heights of both pictures. In the area

where both pictures have pixels, use the pixels from the first picture. However, when those pixels are green, use the pixels from the second picture.

- **E4.23** Using the `Picture` class from Worked Example 4.2, apply a sunset effect to a picture, increasing the red value of each pixel by 30 percent (up to a maximum of 255).
- ■ **E4.24** Using the `Picture` class from Worked Example 4.2, apply a “telescope” effect, turning all pixels black that are outside a circle. The center of the circle should be the image center, and the radius should be 40 percent of the width or height, whichever is smaller.



Cay Horstmann.

- **Graphics E4.25** Write a program to plot the following face.



- **Graphics E4.26** Write a graphical application that displays a checkerboard with 64 squares, alternating white and black.

## PROGRAMMING PROJECTS

- ■ **P4.1** Enhance Worked Example 4.1 to check that the credit card number is valid. A valid credit card number will yield a result divisible by 10 when you:  
Form the sum of all digits. Add to that sum every second digit, starting with the second digit from the right. Then add the number of digits in the second step that are greater than four. The result should be divisible by 10.  
For example, consider the number 4012 8888 8888 1881. The sum of all digits is 89. The sum of the colored digits is 46. There are five colored digits larger than four, so the result is 140. 140 is divisible by 10 so the card number is valid.
- ■ **P4.2** *Mean and standard deviation.* Write a program that reads a set of floating-point data values. Choose an appropriate mechanism for prompting for the end of the data set. When all values have been read, print out the count of the values, the average, and the standard deviation. The average of a data set  $\{x_1, \dots, x_n\}$  is  $\bar{x} = \sum x_i / n$ , where  $\sum x_i = x_1 + \dots + x_n$  is the sum of the input values. The standard deviation is

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

However, this formula is not suitable for the task. By the time the program has computed  $\bar{x}$ , the individual  $x_i$  are long gone. Until you know how to save these values, use the numerically less stable formula

$$s = \sqrt{\frac{\sum x_i^2 - \frac{1}{n}(\sum x_i)^2}{n-1}}$$

You can compute this quantity by keeping track of the count, the sum, and the sum of squares as you process the input values.

- P4.3 The *Fibonacci numbers* are defined by the sequence

$$\begin{aligned}f_1 &= 1 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2}\end{aligned}$$

Reformulate that as

```
fold1 = 1;
fold2 = 1;
fnew = fold1 + fold2;
```

After that, discard `fold2`, which is no longer needed, and set `fold2` to `fold1` and `fold1` to `fnew`. Repeat an appropriate number of times.

Implement a program that prompts the user for an integer  $n$  and prints the  $n$ th Fibonacci number, using the above algorithm.



*Fibonacci numbers describe the growth of a rabbit population.*

© GlobalP/iStockphoto.

- P4.4 *Factoring of integers.* Write a program that asks the user for an integer and then prints out all its factors. For example, when the user enters 150, the program should print

```
2
3
5
5
```

- P4.5 *Prime numbers.* Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print

```
2
3
5
7
11
13
17
19
```

Recall that a number is a prime number if it is not divisible by any number except 1 and itself.

- P4.6 Following Section 4.9, develop a program that reads text and displays the average number of words in each sentence. Assume words are separated by spaces, and a sentence ends when a word ends in a period. Start small and just print the first word. Then print the first two words. Then print all words in the first sentence. Then print the number of words in the first sentence. Then print the number of words

in the first two sentences. Then print the average number of words in the first two sentences. At this time, you should have gathered enough experience that you can complete the program.

- **P4.7** Following Section 4.9, develop a program that reads a string and removes all duplicates. For example, if the input is `Mississippi`, print `Misp`. Start small and just print the first letter. Then print the first letter and true if the letter is not duplicated elsewhere, `false` otherwise. (Look for it in the remaining string, by using the `substring` and `indexOf` methods.) Next, do the same for the first two letters, and print out for each letter whether or not they occur in the substring before and after the letter. Try with a string like `oops`. Extend to all characters in the string. Have a look at the output when the input is `Mississippi`. Which characters should you not report? At this time, you should have gathered enough experience that you can complete the program.

- **P4.8** *The game of Nim.* This is a well-known game with a number of variants. The following variant has an interesting winning strategy. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble loses.

Write a program in which the computer plays against a human opponent. Generate a random integer between 10 and 100 to denote the initial size of the pile. Generate a random integer between 0 and 1 to decide whether the computer or the human takes the first turn. Generate a random integer between 0 and 1 to decide whether the computer plays *smart* or *stupid*. In stupid mode the computer simply takes a random legal value (between 1 and  $n/2$ ) from the pile whenever it has a turn. In smart mode the computer takes off enough marbles to make the size of the pile a power of two minus 1—that is, 3, 7, 15, 31, or 63. That is always a legal move, except when the size of the pile is currently one less than a power of two. In that case, the computer makes a random legal move.

You will note that the computer cannot be beaten in smart mode when it has the first move, unless the pile size happens to be 15, 31, or 63. Of course, a human player who has the first turn and knows the winning strategy can win against the computer.

- **P4.9** *The Drunkard's Walk.* A drunkard in a grid of streets randomly picks one of four directions and stumbles to the next intersection, then again randomly picks one of four directions, and so on. You might think that on average the drunkard doesn't move very far because the choices cancel each other out, but that is actually not the case.

Represent locations as integer pairs  $(x, y)$ . Implement the drunkard's walk over 100 intersections, starting at  $(0, 0)$ , and print the ending location.

- **P4.10** *The Monty Hall Paradox.* Marilyn vos Savant described the following problem (loosely based on a game show hosted by Monty Hall) in a popular magazine: "Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?"

Ms. vos Savant proved that it is to your advantage, but many of her readers, including some mathematics professors, disagreed, arguing that the probability would not change because another door was opened.

Your task is to simulate this game show. In each iteration, randomly pick a door number between 1 and 3 for placing the car. Randomly have the player pick a door. Randomly have the game show host pick a door having a goat (but not the door that the player picked). Increment a counter for strategy 1 if the player wins by switching to the host's choice, and increment a counter for strategy 2 if the player wins by sticking with the original choice. Run 1,000 iterations and print both counters.

- **P4.11** A simple random generator is obtained by the formula

$$r_{\text{new}} = (a \cdot r_{\text{old}} + b) \% m$$

and then setting  $r_{\text{old}}$  to  $r_{\text{new}}$ . If  $m$  is chosen as  $2^{32}$ , then you can compute

$$r_{\text{new}} = a \cdot r_{\text{old}} + b$$

because the truncation of an overflowing result to the `int` type is equivalent to computing the remainder.

Write a program that asks the user to enter a seed value for  $r_{\text{old}}$ . (Such a value is often called a *seed*). Then print the first 100 random integers generated by this formula, using  $a = 32310901$  and  $b = 1729$ .

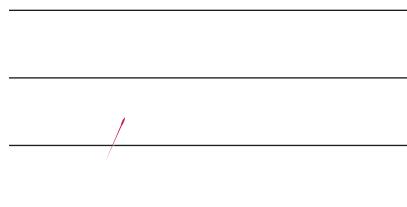
- ■ **P4.12** *The Buffon Needle Experiment.* The following experiment was devised by Comte Georges-Louis Leclerc de Buffon (1707–1788), a French naturalist. A needle of length 1 inch is dropped onto paper that is ruled with lines 2 inches apart. If the needle drops onto a line, we count it as a *hit*. (See Figure 9.) Buffon discovered that the quotient *tries/hits* approximates  $\pi$ .

For the Buffon needle experiment, you must generate two random numbers: one to describe the starting position and one to describe the angle of the needle with the  $x$ -axis. Then you need to test whether the needle touches a grid line.

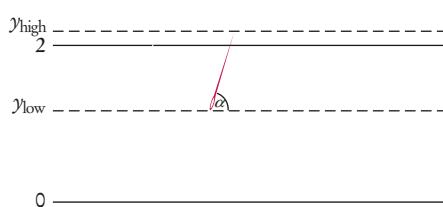
Generate the *lower* point of the needle. Its  $x$ -coordinate is irrelevant, and you may assume its  $y$ -coordinate  $y_{\text{low}}$  to be any random number between 0 and 2. The angle  $\pi$  between the needle and the  $x$ -axis can be any value between 0 degrees and 180 degrees ( $\pi$  radians). The upper end of the needle has  $y$ -coordinate

$$y_{\text{high}} = y_{\text{low}} + \sin \alpha$$

The needle is a hit if  $y_{\text{high}}$  is at least 2, as shown in Figure 10. Stop after 10,000 tries and print the quotient *tries/hits*. (This program is not suitable for computing the value of  $\pi$ . You need  $\pi$  in the computation of the angle.)



**Figure 9**  
The Buffon Needle Experiment



**Figure 10**  
A Hit in the Buffon Needle Experiment

**■■ P4.13** In the 17th century, the discipline of probability theory got its start when a gambler asked a mathematician friend to explain some observations about dice games. Why did he, on average, win a bet that at least one six would appear when rolling a die four times? And why did he seem to lose a similar bet, getting at least one double-six when rolling a pair of dice 24 times?

Nowadays, it seems astounding that any person would roll a pair of dice 24 times in a row, and then repeat that many times over. Let's do that experiment on a computer instead. Simulate each game a million times and print out the wins and losses, assuming each bet was for \$1.

**■■ P4.14** Write a program that reads an integer  $n$  and a digit  $d$  between 0 and 9. Use one or more loops to count how many of the integers between 1 and  $n$

- start with the digit  $d$ .
- end with the digit  $d$ .
- contain the digit  $d$ .

**■ Business P4.15** Write a program that reads an initial investment balance and an interest rate, then prints the number of years it takes for the investment to reach one million dollars.

**■ Business P4.16** *Currency conversion.* Write a program that first asks the user to type today's price for one dollar in Japanese yen, then reads U.S. dollar values and converts each to yen. Use 0 as a sentinel.

**■ Business P4.17** Write a program that first asks the user to type in today's price of one dollar in Japanese yen, then reads U.S. dollar values and converts each to Japanese yen. Use 0 as the sentinel value to denote the end of dollar inputs. Then the program reads a sequence of yen amounts and converts them to dollars. The second sequence is terminated by another zero value.

**■ Business P4.18** Your company has shares of stock it would like to sell when their value exceeds a certain target price. Write a program that reads the target price and then reads the current stock price until it is at least the target price. Your program should use a Scanner to read a sequence of double values from standard input. Once the minimum is reached, the program should report that the stock price exceeds the target price.

**■ Business P4.19** Write an application to pre-sell a limited number of cinema tickets. Each buyer can buy as many as 4 tickets. No more than 100 tickets can be sold. Implement a program called `TicketSeller` that prompts the user for the desired number of tickets and then displays the number of remaining tickets. Repeat until all tickets have been sold, and then display the total number of buyers.

**■ Business P4.20** You need to control the number of people who can be in an oyster bar at the same time. Groups of people can always leave the bar, but a group cannot enter the bar if they would make the number of people in the bar exceed the maximum of 100 occupants. Write a program that reads the sizes of the groups that arrive or depart. Use negative numbers for departures. After each input, display the current number of occupants. As soon as the bar holds the maximum number of people, report that the bar is full and exit the program.

|  |                  |     |        |         |
|--|------------------|-----|--------|---------|
|  | <b>CANADA</b>    | CAD | 0.9512 | 0.8883  |
|  | <b>CHINA</b>     | CNY | 7.3169 | 6.0910  |
|  | <b>EURO</b>      | EUR | 0.6644 | 0.6100  |
|  | <b>JAPAN</b>     | JPY | 109.00 | 102.00  |
|  | <b>SINGAPORE</b> | SGD | 1.3732 | 1.2630  |
|  | <b>HONG KONG</b> | HKD | 10.043 | 8.4072  |
|  |                  |     | 10.646 | 9.10675 |

© hatman12/Stockphoto

**Business P4.21** *Credit Card Number Check.* The last digit of a credit card number is the *check digit*, which protects against transcription errors such as an error in a single digit or switching two digits. The following method is used to verify actual credit card numbers but, for simplicity, we will describe it for numbers with 8 digits instead of 16:

- Starting from the rightmost digit, form the sum of every other digit. For example, if the credit card number is 4358 9795, then you form the sum  $5 + 7 + 8 + 3 = 23$ .
- Double each of the digits that were not included in the preceding step. Add all digits of the resulting numbers. For example, with the number given above, doubling the digits, starting with the next-to-last one, yields 18 18 10 8. Adding all digits in these values yields  $1 + 8 + 1 + 8 + 1 + 0 + 8 = 27$ .
- Add the sums of the two preceding steps. If the last digit of the result is 0, the number is valid. In our case,  $23 + 27 = 50$ , so the number is valid.

Write a program that implements this algorithm. The user should supply an 8-digit number, and you should print out whether the number is valid or not. If it is not valid, you should print the value of the check digit that would make it valid.

**Science P4.22** In a predator-prey simulation, you compute the populations of predators and prey, using the following equations:

$$\begin{aligned} \text{prey}_{n+1} &= \text{prey}_n \times (1 + A - B \times \text{pred}_n) \\ \text{pred}_{n+1} &= \text{pred}_n \times (1 - C + D \times \text{prey}_n) \end{aligned}$$

Here,  $A$  is the rate at which prey birth exceeds natural death,  $B$  is the rate of predation,  $C$  is the rate at which predator deaths exceed births without food, and  $D$  represents predator increase in the presence of food.

Write a program that prompts users for these rates, the initial population sizes, and the number of periods. Then print the populations for the given number of periods. As inputs, try  $A = 0.1$ ,  $B = C = 0.01$ , and  $D = 0.00002$  with initial prey and predator populations of 1,000 and 20.



© Charles Gibson/iStockphoto.

**Science P4.23** *Projectile flight.* Suppose a cannonball is propelled straight into the air with a starting velocity  $v_0$ . Any calculus book will state that the position of the ball after  $t$  seconds is  $s(t) = -\frac{1}{2}gt^2 + v_0t$ , where  $g = 9.81 \text{ m/s}^2$  is the gravitational force of the earth. No calculus textbook ever mentions why someone would want to carry out such an obviously dangerous experiment, so we will do it in the safety of the computer.

In fact, we will confirm the theorem from calculus by a simulation. In our simulation, we will consider how the ball moves in very short time intervals  $\Delta t$ . In a short time interval the velocity  $v$  is nearly constant, and we can compute the distance the ball moves as  $\Delta s = v\Delta t$ . In our program, we will simply set

```
final double DELTA_T = 0.01;
```

and update the position by

```
s = s + v * DELTA_T;
```



© MOFI/iStockphoto.

The velocity changes constantly—in fact, it is reduced by the gravitational force of the earth. In a short time interval,  $\Delta v = -g\Delta t$ , we must keep the velocity updated as

$$v = v - g * \text{DELTA\_T};$$

In the next iteration the new velocity is used to update the distance.

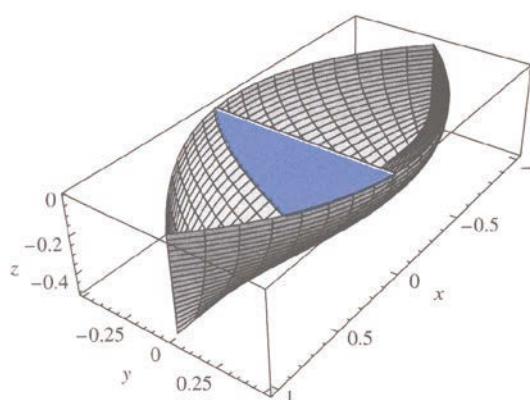
Now run the simulation until the cannonball falls back to the earth. Get the initial velocity as an input (100 m/s is a good value). Update the position and velocity 100 times per second, but print out the position only every full second. Also printout the values from the exact formula  $s(t) = -\frac{1}{2}gt^2 + v_0t$  for comparison.

*Note:* You may wonder whether there is a benefit to this simulation when an exact formula is available. Well, the formula from the calculus book is *not* exact. Actually, the gravitational force diminishes the farther the cannonball is away from the surface of the earth. This complicates the algebra sufficiently that it is not possible to give an exact formula for the actual motion, but the computer simulation can simply be extended to apply a variable gravitational force. For cannonballs, the calculus-book formula is actually good enough, but computers are necessary to compute accurate trajectories for higher-flying objects such as ballistic missiles.

**\*\*\* Science P4.24** A simple model for the hull of a ship is given by

$$|y| = \frac{B}{2} \left[ 1 - \left( \frac{2x}{L} \right)^2 \right] \left[ 1 - \left( \frac{z}{T} \right)^2 \right]$$

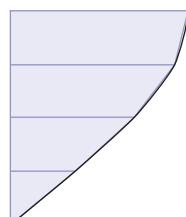
where  $B$  is the beam,  $L$  is the length, and  $T$  is the draft. (*Note:* There are two values of  $y$  for each  $x$  and  $z$  because the hull is symmetric from starboard to port.)



(left and below) Courtesy of James P. Holloway/  
John Wiley & Sons, Inc.

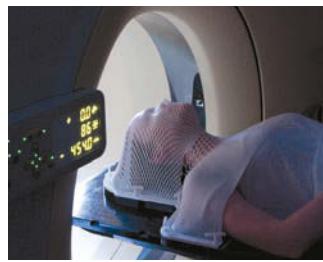
The cross-sectional area at a point  $x$  is called the “section” in nautical parlance. To compute it, let  $z$  go from 0 to  $-T$  in  $n$  increments, each of size  $T/n$ . For each value of  $z$ , compute the value for  $y$ . Then sum the areas of trapezoidal strips. At right are the strips where  $n = 4$ .

Write a program that reads in values for  $B$ ,  $L$ ,  $T$ ,  $x$ , and  $n$  and then prints out the cross-sectional area at  $x$ .



**■ Science P4.25** Radioactive decay of radioactive materials can be modeled by the equation  $A = A_0 e^{-t(\log 2/b)}$ , where  $A$  is the amount of the material at time  $t$ ,  $A_0$  is the amount at time 0, and  $b$  is the half-life.

Technetium-99 is a radioisotope that is used in imaging of the brain. It has a half-life of 6 hours. Your program should display the relative amount  $A / A_0$  in a patient body every hour for 24 hours after receiving a dose.



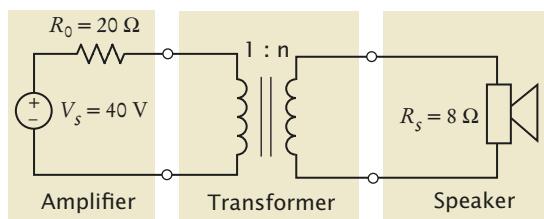
© Snowleopard1/iStock photo.

**■■■ Science P4.26**

The photo at left shows an electric device called a “transformer”. Transformers are often constructed by wrapping coils of wire around a ferrite core. The figure below illustrates a situation that occurs in various audio devices such as cell phones and music players. In this circuit, a transformer is used to connect a speaker to the output of an audio amplifier.



© zig4photo/iStock photo.



The symbol used to represent the transformer is intended to suggest two coils of wire. The parameter  $n$  of the transformer is called the “turns ratio” of the transformer. (The number of times that a wire is wrapped around the core to form a coil is called the number of turns in the coil. The turns ratio is literally the ratio of the number of turns in the two coils of wire.)

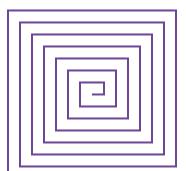
When designing the circuit, we are concerned primarily with the value of the power delivered to the speakers—that power causes the speakers to produce the sounds we want to hear. Suppose we were to connect the speakers directly to the amplifier without using the transformer. Some fraction of the power available from the amplifier would get to the speakers. The rest of the available power would be lost in the amplifier itself. The transformer is added to the circuit to increase the fraction of the amplifier power that is delivered to the speakers.

The power,  $P_s$ , delivered to the speakers is calculated using the formula

$$P_s = R_s \left( \frac{nV_s}{n^2R_0 + R_s} \right)^2$$

Write a program that models the circuit shown and varies the turns ratio from 0.01 to 2 in 0.01 increments, then determines the value of the turns ratio that maximizes the power delivered to the speakers.

**■■■ Graphics P4.27** Write a graphical application that draws a spiral, such as the following:



**■■ Graphics P4.28** It is easy and fun to draw graphs of curves with the Java graphics library. Simply draw 100 line segments joining the points  $(x, f(x))$  and  $(x + d, f(x + d))$ , where  $x$  ranges from  $x_{\min}$  to  $x_{\max}$  and  $d = (x_{\max} - x_{\min})/100$ .

Draw the curve  $f(x) = 0.00005x^3 - 0.03x^2 + 4x + 200$ , where  $x$  ranges from 0 to 400 in this fashion.

**■■■ Graphics P4.29** Draw a picture of the “four-leaved rose” whose equation in polar coordinates is  $r = \cos(2\theta)$ . Let  $\theta$  go from 0 to  $2\pi$  in 100 steps. Each time, compute  $r$  and then compute the  $(x, y)$  coordinates from the polar coordinates by using the formula

$$x = r \cdot \cos(\theta), y = r \cdot \sin(\theta)$$

## ANSWERS TO SELF-CHECK QUESTIONS

**1.** 23 years.

**2.** 8 years.

**3.** Add a statement

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

as the last statement in the `while` loop.

**4.** The program prints the same output. This is because the balance after 14 years is slightly below \$20,000, and after 15 years, it is slightly above \$20,000.

**5.** 2 4 8 16 32 64 128

Note that the value 128 is printed even though it is larger than 100.

**6.** n output

|    |    |
|----|----|
| 5  |    |
| 4  | 4  |
| 3  | 3  |
| 2  | 2  |
| 1  | 1  |
| 0  | 0  |
| -1 | -1 |

**7.** n output

|   |          |
|---|----------|
| 1 | 1,       |
| 2 | 1, 2,    |
| 3 | 1, 2, 3, |
| 4 |          |

There is a comma after the last value. Usually, commas are between values only.

**8.** a n r i

|   |   |    |   |
|---|---|----|---|
| 2 | 4 | 1  | 1 |
|   |   | 2  | 2 |
|   |   | 4  | 3 |
|   |   | 8  | 4 |
|   |   | 16 | 5 |

The code computes  $a^n$ .

**9.** n output

|     |    |
|-----|----|
| 1   | 1  |
| 11  | 11 |
| 21  | 21 |
| 31  | 31 |
| 41  | 41 |
| 51  | 51 |
| 61  | 61 |
| ... |    |

This is an infinite loop. `n` is never equal to 50.

**10.** count temp

|   |      |
|---|------|
| 1 | 123  |
| 2 | 12.3 |
| 3 | 1.23 |

This yields the correct answer. The number 123 has 3 digits.

count temp

|   |      |
|---|------|
| 1 | 100  |
| 2 | 10.0 |

This yields the wrong answer. The number 100 also has 3 digits. The loop condition should have been

`while (temp >= 10)`

**11.** int year = 1;

while (year <= nyears)

```
{
 double interest = balance * RATE / 100;
 balance = balance + interest;
 System.out.printf("%4d %10.2f%n",
 year, balance);
 year++;
}
```

**12.** 11 numbers: 10 9 8 7 6 5 4 3 2 1 0

**13.** for (int i = 10; i <= 20; i = i + 2)

{

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

**14.** int sum = 0;  
 for (int i = 1; i <= n; i++)  
 {  
 sum = sum + i;  
}

**15.** for (int year = 1;  
 balance <= 2 \* INITIAL\_BALANCE; year++)

However, it is best not to use a for loop in this case because the loop condition does not relate to the year variable. A while loop would be a better choice.

**16.** do  
{  
 System.out.print(  
 "Enter a value between 0 and 100: ");  
 value = in.nextInt();  
}  
while (value < 0 || value > 100);

**17.** int value = 100;  
while (value >= 100)  
{  
 System.out.print("Enter a value < 100: ");  
 value = in.nextInt();  
}

Here, the variable value had to be initialized with an artificial value to ensure that the loop is entered at least once.

**18.** Yes. The do loop

```
do { body } while (condition);
```

is equivalent to this while loop:

```
boolean first = true;

while (first || condition)

{

 body;

 first = false;

}
```

**19.** int x;  
int sum = 0;  
do  
{  
 x = in.nextInt();  
 sum = sum + x;  
}  
while (x != 0);

**20.** int x = 0;  
int previous;  
do  
{  
 previous = x;  
 x = in.nextInt();  
 sum = sum + x;  
}

```
}

while (x != 0 && previous != x);
```

**21.** No data

**22.** The first check ends the loop after the sentinel has been read. The second check ensures that the sentinel is not processed as an input value.

**23.** The while loop would never be entered. The user would never be prompted for input. Because count stays 0, the program would then print "No data".

**24.** The hasNextDouble method also returns false. A more accurate prompt would have been: "Enter values, a key other than a digit to quit." But that might be more confusing to the program user who would need to ponder which key to choose.

**25.** If the user doesn't provide any numeric input, the first call to in.nextDouble() will fail.

#### 26. Computing the average

```
Enter scores, Q to quit: 90 80 90 100 80 Q

The average is 88

(Program exits)
```

#### 27. Simple conversion

```
Your conversion question: How many in are 30 cm

30 cm = 11.81 in

(Program exits) Only one value can be converted

Run program again for another question
```

#### Unknown unit

```
Your conversion question: How many inches are 30 cm?

Unknown unit: inches

Known units are in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal

(Program exits)
```

#### Program doesn't understand question syntax

```
Your conversion question: What is an ångström?

Please formulate your question as "How many (unit) are (value) (unit)?"

(Program exits)
```

#### 28. One score is not enough

```
Enter scores, Q to quit: 90 Q

Error: At least two scores are required.

(Program exits)
```

**29.** It would not be possible to implement this interface using the Java features we have covered up to this point. There is no way for the program to know when the first set of inputs

ends. (When you read numbers with `value = in.nextDouble()`, it is your choice whether to put them on a single line or multiple lines.)

**30. Comparing two interest rates**

First interest rate in percent: 5  
Second interest rate in percent: 10

Years: 5

| Year | 5%       | 10%      |
|------|----------|----------|
| 0    | 10000.00 | 10000.00 |
| 1    | 10500.00 | 11000.00 |
| 2    | 11025.00 | 12100.00 |
| 3    | 11576.25 | 13310.00 |
| 4    | 12155.06 | 14641.00 |
| 5    | 12762.82 | 16105.10 |

This row clarifies that 1 means the end of the first year

**31.** The total is zero.

**32.** `double total = 0;`  
`while (in.hasNextDouble())`  
`{`  
 `double input = in.nextDouble();`  
 `if (input > 0) { total = total + input; }`  
`}`

**33.** `position` is `str.length()` and `ch` is set to the last character of the string or, if the string is empty, is unchanged from its initial value, '?'. Note that `ch` must be initialized with some value—otherwise the compiler will complain about a possibly uninitialized variable.

**34.** The loop will stop when a match is found, but you cannot access the match because neither `position` nor `ch` are defined outside the loop.

**35.** Start the loop at the end of string:

```
boolean found = false;
int i = str.length() - 1;
while (!found && i >= 0)
{
 char ch = str.charAt(i);
 if (ch == ' ') { found = true; }
 else { i--; }
}
```

**36.** The initial call to `in.nextDouble()` fails, terminating the program. One solution is to do all input in the loop and introduce a Boolean variable that checks whether the loop is entered for the first time.

```
double input = 0;
boolean first = true;
while (in.hasNextDouble())
{
 double previous = input;
 input = in.nextDouble();
```

```
if (first) { first = false; }
else if (input == previous)
{
 System.out.println("Duplicate input");
}
```

**37.** All values in the inner loop should be displayed on the same line.

**38.** Change lines 13, 18, and 30 to `for (int n = 0; n <= NMAX; n++)`. Change `NMAX` to 5.

**39.** 60: The outer loop is executed 10 times, and the inner loop 6 times.

**40.** 0123  
1234  
2345

**41.** `for (int i = 1; i <= 3; i++)`  
`{`  
 `for (int j = 1; j <= 4; j++)`  
 `{`  
 `System.out.print("[]");`  
 `}`  
 `System.out.println();`  
`}`

**42.** Of course, there is more than one way to simplify the problem. One way is to print the words in which the first letter is not repeated.

**43.** You could first write a program that prints all substrings of a given string.

**44.** You can look for a single red pixel, or a block of nine neighboring red pixels.

**45.** Here is one plan:

- Find the position of the first digit in a string.
- Find the position of the first non-digit after a given position in a string.
- Extract the first integer from a string (using the preceding two steps).
- Print all integers from a string. (Use the first three steps, then repeat with the substring that starts after the extracted integer.)

**46.** Compute `(int) (Math.random() * 2)`, and use 0 for heads, 1 for tails, or the other way around.

**47.** Compute `(int) (Math.random() * 4)` and associate the numbers 0...3 with the four suits. Then compute `(int) (Math.random() * 13)` and associate the numbers 0...12 with Jack, Ace, 2...10, Queen, and King.

- 48.** We need to call it once for each die. If we printed the same value twice, the die tosses would not be independent.
- 49.** The call will produce a value between 2 and 12, but all values have the same probability. When throwing a pair of dice, the number 7 is six times as likely as the number 2. The correct formula is

```
int sum = (int) (Math.random() * 6) + (int)
 (Math.random() * 6) + 2;
```

- 50.** `Math.random() * 100.0`

## WORKED EXAMPLE 4.1

**Credit Card Processing**

One of the minor annoyances of online shopping is that many web sites require you to enter a credit card without spaces or dashes, which makes double-checking the number rather tedious. How hard can it be to remove dashes or spaces from a string? Not hard at all, as this worked example shows.

**Problem Statement** Your task is to remove all spaces or dashes from a string creditCardNumber. For example, if creditCardNumber is “4123-5678-9012-3450”, then you should set it to “4123567890123450”.

**Credit Card Information (all fields are required)**

|                     |                                                              |  |  |
|---------------------|--------------------------------------------------------------|--|--|
| We Accept:          |                                                              |  |  |
| Credit Card Type:   | <input type="button" value="▼"/>                             |  |  |
| Credit Card Number: | <input type="text"/> <i>(Do not enter spaces or dashes.)</i> |  |  |

**Step 1** Decide what work must be done *inside* the loop.

In the loop, we visit each character in turn. You can get the *i*th character as

```
char ch = creditCardNumber.charAt(i);
```

If it is not a dash or space, we move on to the next character. If it is a dash or space, we remove the offending character.

*i = 0*

**While ...**

Set *ch* to the *i*th character of creditCardNumber.

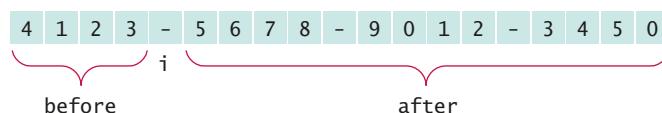
If *ch* is a space or dash

Remove the character from creditCardNumber.

**Else**

**Increment i.**

You may wonder how to remove a character from a string in Java. Here is the procedure for removing the character at position *i*: Take the substrings that end before *i* and start after *i*, and concatenate them.



```
String before = creditCardNumber.substring(0, i);
String after = creditCardNumber.substring(i + 1);
creditCardNumber = before + after;
```

Note that we do *not* increment *i* after removing a character. For example, in the figure above, *i* was 4, and we removed the dash at position 4. The next time we enter the loop, we want to reexamine position 4 which now contains the character 5.

**Step 2** Specify the loop condition.

We stay in the loop while the index *i* is a valid position. That is,

```
i < creditCardNumber.length()
```

**Step 3** Choose the loop type.

We don't know at the outset how often the loop is repeated. It depends on the number of dashes and spaces that we find. Therefore, we will choose a `while` loop. Why not a `do` loop? If we are given an empty string (because the user has not provided any credit card number at all), we do not want to enter the loop at all.

**Step 4** Process the result after the loop has finished.

In this case, the result is simply the string.

**Step 5** Trace the loop with typical examples.

The complete loop is

```
i = 0
While i < creditCardNumber.length()
 ch = the ith character of creditCardNumber.
 If ch is a space or dash
 Remove the character from creditCardNumber.
 Else
 Increment i.
```

It is a bit tedious to trace a string with 20 characters, so we will use a shorter example:

| creditCardNumber | i | ch |
|------------------|---|----|
| 4-56-7           | 0 | 4  |
| 4-56-7           | 1 | -  |
| 456-7            | 1 | 5  |
| 456-7            | 2 | 6  |
| 456-7            | 3 | -  |
| 4567             | 3 | 7  |

**Step 6** Implement the loop in Java.

Here's the complete program:

**worked\_example\_1/CCNumber.java**

```
1 /**
2 * This program removes spaces and dashes from a credit card number.
3 */
4 public class CCNumber
5 {
6 public static void main(String[] args)
7 {
8 String creditCardNumber = "4123-5678-9012-3450";
9
10 int i = 0;
11 while (i < creditCardNumber.length())
12 {
13 char ch = creditCardNumber.charAt(i);
14 if (ch == ' ' || ch == '-')
15 {
```

```
16 // Remove the character at position i
17
18 String before = creditCardNumber.substring(0, i);
19 String after = creditCardNumber.substring(i + 1);
20 creditCardNumber = before + after;
21 }
22 else
23 {
24 i++;
25 }
26 }
27
28 System.out.println(creditCardNumber);
29 }
30 }
```



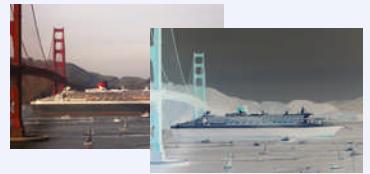
## WORKED EXAMPLE 4.2

## Manipulating the Pixels in an Image



A digital image is made up of *pixels*. Each pixel is a tiny square of a given color. In this Worked Example, we will use a class `Picture` that has methods for loading an image and accessing its pixels.

**Problem Statement** Your task is to convert an image into its negative, turning white to black, cyan to red, and so on. The result is a negative image of the kind that old-fashioned film cameras used to produce.



Cay Horstmann.

The implementation of the `Picture` class uses the Java image library and is beyond the scope of this book, but here are the relevant parts of the public interface:

```
public class Picture
{
 ...
 /**
 * Gets the width of this picture.
 * @return the width
 */
 public int getWidth() { . . . }

 /**
 * Gets the height of this picture.
 * @return the height
 */
 public int getHeight() { . . . }

 /**
 * Loads a picture from a given source.
 * @param source the image source. If the source starts
 * with http://, it is a URL, otherwise, a filename.
 */
 public void load(String source) { . . . }

 /**
 * Gets the color of a pixel.
 * @param x the column index (between 0 and getWidth() - 1)
 * @param y the row index (between 0 and getHeight() - 1)
 * @return the color of the pixel at position (x, y)
 */
 public Color getColorAt(int x, int y) { . . . }

 /**
 * Sets the color of a pixel.
 * @param x the column index (between 0 and getWidth() - 1)
 * @param y the row index (between 0 and getHeight() - 1)
 * @param c the color for the pixel at position (x, y)
 */
 public void setColorAt(int x, int y, Color c) { . . . }

 ...
}
```

Now consider the task of converting an image into its negative.



Cay Horstmann.

The negative of a `Color` object is computed like this:

```
Color original = ...;
Color negative = new Color(255 - original.getRed(),
 255 - original.getGreen(),
 255 - original.getBlue());
```

We want to apply this operation to each pixel in the image.

To process all pixels, we can use one of the following two strategies:

**For each row**  
**For each pixel in the row**  
**Process the pixel.**

or

**For each column**  
**For each pixel in the column**  
**Process the pixel.**

Because our pixel class uses *x/y* coordinates to access a pixel, it turns out to be more natural to use the second strategy. (In Chapter 6, you will encounter two-dimensional arrays that are accessed with row/column coordinates. In that situation, use the first form.)

To traverse each column, the *x*-coordinate starts at 0. Since there are `pic.getWidth()` columns, we use the loop

```
for (int x = 0; x < pic.getWidth(); x++)
```

Once a column has been fixed, we need to traverse all *y*-coordinates in that column, starting from 0. There are `pic.getHeight()` rows, so our nested loops are

```
for (int x = 0; x < pic.getWidth(); x++)
{
 for (int y = 0; y < pic.getHeight(); y++)
 {
 Color original = pic.getColorAt(x, y);
 ...
 }
}
```

The following program solves our image manipulation problem:

### worked\_example\_2/Negative.java

```
1 import java.awt.Color;
2
3 public class Negative
4 {
5 public static void main(String[] args)
6 {
```

```
7 Picture pic = new Picture();
8 pic.load("queen-mary.png");
9 for (int x = 0; x < pic.getWidth(); x++)
10 {
11 for (int y = 0; y < pic.getHeight(); y++)
12 {
13 Color original = pic.getColorAt(x, y);
14 Color negative = new Color(255 - original.getRed(),
15 255 - original.getGreen(),
16 255 - original.getBlue());
17 pic.setColorAt(x, y, negative);
18 }
19 }
20 }
21 }
```



# METHODS

## CHAPTER GOALS

- To be able to implement methods
- To become familiar with the concept of parameter passing
- To develop strategies for decomposing complex tasks into simpler ones
- To be able to determine the scope of a variable
- To learn how to think recursively (optional)



© attattor/iStockphoto.

## CHAPTER CONTENTS

|                                                 |     |
|-------------------------------------------------|-----|
| <b>5.1 METHODS AS BLACK BOXES</b>               | 212 |
| <b>5.2 IMPLEMENTING METHODS</b>                 | 214 |
| <b>SYN</b> Static Method Declaration            | 215 |
| <b>PT1</b> Method Comments                      | 217 |
| <b>5.3 PARAMETER PASSING</b>                    | 217 |
| <b>PT2</b> Do Not Modify Parameter Variables    | 219 |
| <b>CE1</b> Trying to Modify Arguments           | 219 |
| <b>5.4 RETURN VALUES</b>                        | 220 |
| <b>CE2</b> Missing Return Value                 | 222 |
| <b>HT1</b> Implementing a Method                | 222 |
| <b>WE1</b> Generating Random Passwords          | 223 |
| <b>5.5 METHODS WITHOUT RETURN VALUES</b>        | 224 |
| <b>5.6 PROBLEM SOLVING: REUSABLE METHODS</b>    | 225 |
| <b>C&amp;S</b> Personal Computing               | 228 |
| <b>5.7 PROBLEM SOLVING: STEPWISE REFINEMENT</b> | 229 |
| <b>PT3</b> Keep Methods Short                   | 234 |
| <b>PT4</b> Tracing Methods                      | 234 |
| <b>PT5</b> Stubs                                | 235 |
| <b>WE2</b> Calculating a Course Grade           | 236 |
| <b>5.8 VARIABLE SCOPE</b>                       | 236 |
| <b>PT6</b> Using a Debugger                     | 239 |
| <b>VE1</b> Debugging                            | 239 |
| <b>5.9 RECURSIVE METHODS (OPTIONAL)</b>         | 240 |
| <b>HT2</b> Thinking Recursively                 | 243 |
| <b>VE2</b> Fully Justified Text                 | 243 |



© attator/iStockphoto.

A method packages a computation consisting of multiple steps into a form that can be easily understood and reused. (The person in the image to the left is in the middle of executing the method “make espresso”.)

In this chapter, you will learn how to design and implement your own methods. Using the process of stepwise refinement, you will be able to break up complex tasks into sets of cooperating methods.

## 5.1 Methods as Black Boxes

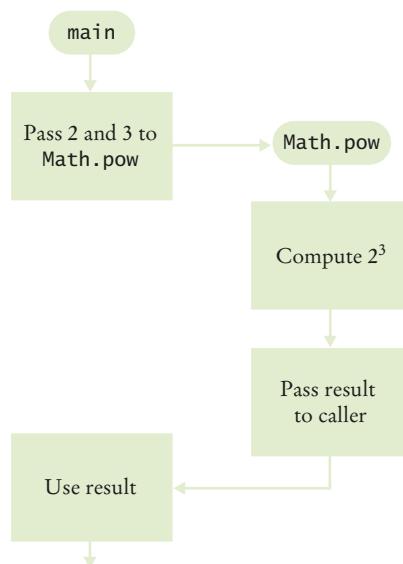
A method is a named sequence of instructions.

A **method** is a sequence of instructions with a name. You have already encountered several methods. For example, the `Math.pow` method, which was introduced in Chapter 2, contains instructions to compute a power  $x^y$ . Moreover, every Java program has a method called `main`.

You *call* a method in order to execute its instructions. For example, consider the following program fragment:

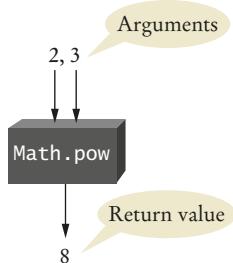
```
public static void main(String[] args)
{
 double result = Math.pow(2, 3);
 ...
}
```

By using the expression `Math.pow(2, 3)`, `main` *calls* the `Math.pow` method, asking it to compute  $2^3$ . The instructions of the `Math.pow` method execute and compute the result. The `Math.pow` method *returns* its result back to `main`, and the `main` method resumes execution (see Figure 1).



**Figure 1** Execution Flow During a Method Call

**Figure 2**  
The Math.pow Method  
as a Black Box



Arguments are supplied when a method is called.

The return value is the result that the method computes.

When another method calls the `Math.pow` method, it provides “inputs”, such as the values 2 and 3 in the call `Math.pow(2, 3)`. These values are called the **arguments** of the method call. Note that they are not necessarily inputs provided by a human user. They are simply the values for which we want the method to compute a result. The “output” that the `Math.pow` method computes is called the **return value**.

Methods can receive multiple arguments, but they return only one value. It is also possible to have methods with no arguments. An example is the `Math.random` method that requires no argument to produce a random number.

The return value of a method is returned to the calling method, where it is processed according to the statement containing the method call. For example, suppose your program contains a statement

```
double result = Math.pow(2, 3);
```

When the `Math.pow` method returns its result, the return value is stored in the variable `result`.

Do not confuse returning a value with producing program output. If you want the return value to be printed, you need to add a statement such as `System.out.print(result)`.

At this point, you may wonder how the `Math.pow` method performs its job. For example, how does `Math.pow` compute that  $2^3$  is 8? By multiplying  $2 \times 2 \times 2$ ? With logarithms? Fortunately, as a user of the method, you *don't need to know* how the method is implemented. You just need to know the *specification* of the method: If you provide arguments  $x$  and  $y$ , the method returns  $x^y$ . Engineers use the term *black box* for a device with a given specification but unknown implementation. You can think of `Math.pow` as a black box, as shown in Figure 2.

When you design your own methods, you will want to make them appear as black boxes to other programmers. Those programmers want to use your methods without knowing what goes on inside. Even if you are the only person working on a program, making each method into a black box pays off: there are fewer details that you need to keep in mind.

Although a thermostat is usually white, you can think of it as a “black box”. The input is the desired temperature, and the output is a signal to the heater or air conditioner.



© yenwen/Stockphoto.



1. Consider the method call `Math.pow(3, 2)`. What are the arguments and return values?
2. What is the return value of the method call `Math.pow(Math.pow(2, 2), 2)`?
3. The `Math.ceil` method in the Java standard library is described as follows: The method receives a single argument  $a$  of type `double` and returns the smallest `double` value  $\geq a$  that is an integer. What is the return value of `Math.ceil(2.3)`?
4. It is possible to determine the answer to Self Check 3 without knowing how the `Math.ceil` method is implemented. Use an engineering term to describe this aspect of the `Math.ceil` method.

**Practice It** Now you can try these exercises at the end of the chapter: R5.3, R5.6.

## 5.2 Implementing Methods

In this section, you will learn how to implement a method from a given specification. We will use a very simple example: a method to compute the volume of a cube with a given side length.



© studioarantanita/iStockphoto.

When declaring a method, you provide a name for the method, a variable for each argument, and a type for the result.

When writing this method, you need to

- Pick a name for the method (`cubeVolume`).
- Declare a variable for each argument (`double sideLength`). These variables are called the **parameter variables**.
- Specify the type of the return value (`double`).
- Add the `public static` modifiers. We will discuss the meanings of these modifiers in Chapter 8. For now, you should simply add them to your methods.

Put all this information together to form the first line of the method's declaration:

```
public static double cubeVolume(double sideLength)
```

This line is called the *header* of the method. Next, specify the **body** of the method. The body contains the variable declarations and statements that are executed when the method is called.

The volume of a cube of side length  $s$  is  $s \times s \times s$ . However, for greater clarity, our parameter variable has been called `sideLength`, not  $s$ , so we need to compute `sideLength * sideLength * sideLength`.

We will store this value in a variable called `volume`:

```
double volume = sideLength * sideLength * sideLength;
```

In order to return the result of the method, use the `return` statement:

```
return volume;
```

*The return statement gives the method's result to the caller.*



© princessdlaif/Stockphoto.

The body of a method is enclosed in braces. Here is the complete method:

```
public static double cubeVolume(double sideLength)
{
 double volume = sideLength * sideLength * sideLength;
 return volume;
}
```

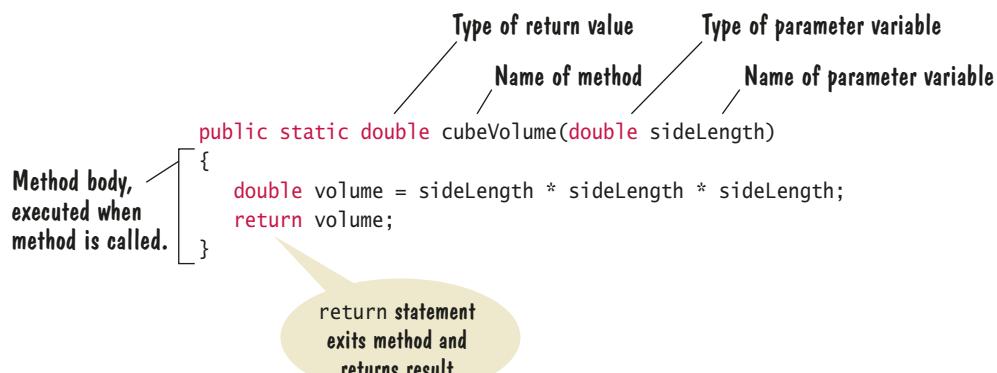
Let's put this method to use. We'll supply a `main` method that calls the `cubeVolume` method twice.

```
public static void main(String[] args)
{
 double result1 = cubeVolume(2);
 double result2 = cubeVolume(10);
 System.out.println("A cube with side length 2 has volume " + result1);
 System.out.println("A cube with side length 10 has volume " + result2);
}
```

When the method is called with different arguments, the method returns different results. Consider the call `cubeVolume(2)`. The argument 2 corresponds to the `sideLength` parameter variable. Therefore, in this call, `sideLength` is 2. The method computes

## Syntax 5.1 Static Method Declaration

**Syntax**    `public static returnType methodName(parameterType parameterName, . . . )`  
`{`  
`method body`  
`}`



`sideLength * sideLength * sideLength`, or  $2 * 2 * 2$ . When the method is called with a different argument, say 10, then the method computes  $10 * 10 * 10$ .

Now we combine both methods into a test program. Note that both methods are contained in the same class. Also note the comment that describes the behavior of the `cubeVolume` method. (Programming Tip 5.1 describes the format of the comment.)

### sec02/Cubes.java

```

1 /**
2 * This program computes the volumes of two cubes.
3 */
4 public class Cubes
5 {
6 public static void main(String[] args)
7 {
8 double result1 = cubeVolume(2);
9 double result2 = cubeVolume(10);
10 System.out.println("A cube with side length 2 has volume " + result1);
11 System.out.println("A cube with side length 10 has volume " + result2);
12 }
13
14 /**
15 * Computes the volume of a cube.
16 * @param sideLength the side length of the cube
17 * @return the volume
18 */
19 public static double cubeVolume(double sideLength)
20 {
21 double volume = sideLength * sideLength * sideLength;
22 return volume;
23 }
24 }
```

### Program Run

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

### SELF CHECK



5. What is the value of `cubeVolume(3)`?
6. What is the value of `cubeVolume(cubeVolume(2))`?
7. Provide an alternate implementation of the body of the `cubeVolume` method by calling the `Math.pow` method.
8. Declare a method `squareArea` that computes the area of a square of a given side length.
9. Consider this method:

```
public static int mystery(int x, int y)
{
 double result = (x + y) / (y - x);
 return result;
}
```

What is the result of the call `mystery(2, 3)`?

**Practice It** Now you can try these exercises at the end of the chapter: R5.1, R5.2, E5.5, E5.14.

## Programming Tip 5.1

**Method Comments**

Whenever you write a method, you should *comment* its behavior. Comments are for human readers, not compilers. The Java language provides a standard layout for method comments, called the **javadoc** convention, as shown here:

```
/***
 * Computes the volume of a cube.
 * @param sideLength the side length of the cube
 * @return the volume
 */
public static double cubeVolume(double sideLength)
{
 double volume = sideLength * sideLength * sideLength;
 return volume;
}
```

Method comments explain the purpose of the method, the meaning of the parameter variables and return value, as well as any special requirements.

Comments are enclosed in `/**` and `*/` delimiters. The first line of the comment describes the purpose of the method. Each `@param` clause describes a parameter variable and the `@return` clause describes the return value.

Note that the method comment does not document the implementation (*how* the method carries out its work) but rather the design (*what* the method does). The comment allows other programmers to use the method as a “black box”.

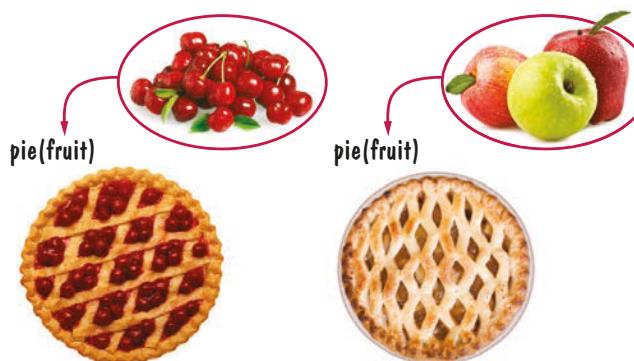
## 5.3 Parameter Passing

Parameter variables hold the arguments supplied in the method call.

In this section, we examine the mechanism of parameter passing more closely. When a method is called, variables are created for receiving the method’s arguments. These variables are called **parameter variables**. (Another commonly used term is **formal parameters**.) The values that are supplied to the method when it is called are the **arguments** of the call. (These values are also commonly called the **actual parameters**.) Each parameter variable is initialized with the corresponding argument.

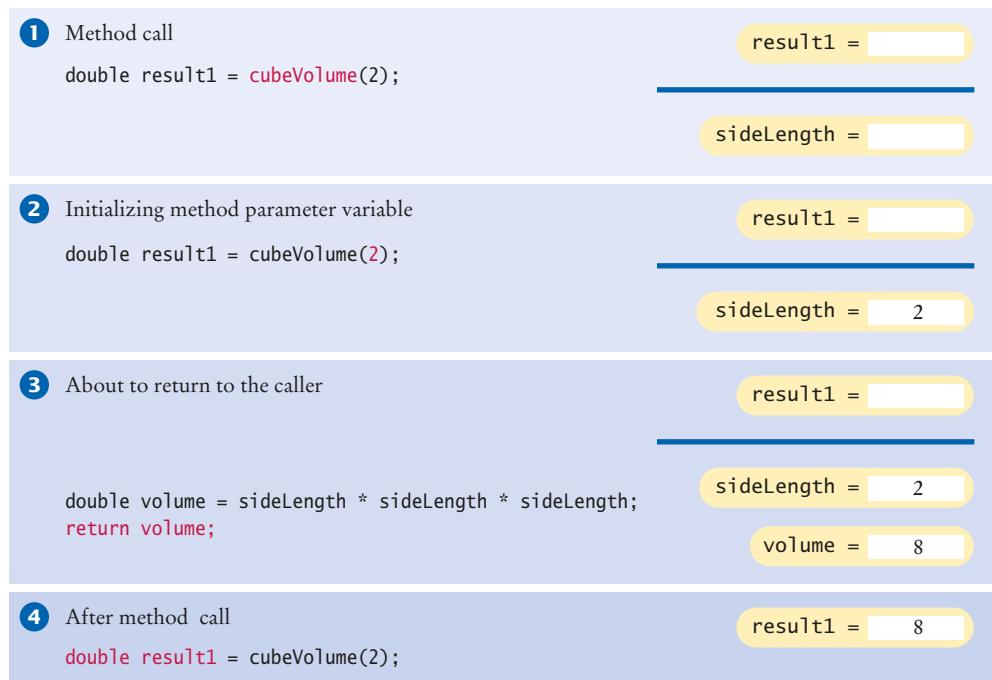
Consider the method call illustrated in Figure 3:

```
double result1 = cubeVolume(2);
```



© DNY59/Stockphoto (cherry pie);  
© inhauscreative/Stockphoto (apple pie);  
© RedHelga/Stockphoto (cherries);  
© ZoneCreative/Stockphoto (apples).

A recipe for a fruit pie may say to use any kind of fruit. Here, “fruit” is an example of a parameter variable. Apples and cherries are examples of arguments.

**Figure 3** Parameter Passing

- The parameter variable `sideLength` of the `cubeVolume` method is created when the method is called. **1**
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, `sideLength` is set to 2. **2**
- The method computes the expression `sideLength * sideLength * sideLength`, which has the value 8. That value is stored in the variable `volume`. **3**
- The method returns. All of its variables are removed. The return value is transferred to the *caller*, that is, the method calling the `cubeVolume` method. The caller puts the return value in the `result1` variable. **4**

Now consider what happens in a subsequent call, `cubeVolume(10)`. A new parameter variable is created. (Recall that the previous parameter variable was removed when the first call to `cubeVolume` returned.) It is initialized with 10, and the process repeats. After the second method call is complete, its variables are again removed.

**SELF CHECK**

10. What does this program print? Use a diagram like Figure 3 to find the answer.

```
public static double mystery(int x, int y)
{
 double z = x + y;
 z = z / 2.0;
 return z;
}
public static void main(String[] args)
{
 int a = 5;
 int b = 7;
```

```

 System.out.println(mystery(a, b));
 }
}

```

- 11.** What does this program print? Use a diagram like Figure 3 to find the answer.

```

public static int mystery(int x)
{
 int y = x * x;
 return y;
}
public static void main(String[] args)
{
 int a = 4;
 System.out.println(mystery(a + 1));
}

```

- 12.** What does this program print? Use a diagram like Figure 3 to find the answer.

```

public static int mystery(int n)
{
 n++;
 n++;
 return n;
}
public static void main(String[] args)
{
 int a = 5;
 System.out.println(mystery(a));
}

```

**Practice It** Now you can try these exercises at the end of the chapter: R5.5, R5.14, P5.1.

### Programming Tip 5.2



#### Do Not Modify Parameter Variables

In Java, a parameter variable is just like any other variable. You can modify the values of the parameter variables in the body of a method. For example,

```

public static int totalCents(int dollars, int cents)
{
 cents = dollars * 100 + cents; // Modifies parameter variable
 return cents;
}

```

However, many programmers find this practice confusing (see Common Error 5.1). To avoid the confusion, simply introduce a separate variable:

```

public static int totalCents(int dollars, int cents)
{
 int result = dollars * 100 + cents;
 return result;
}

```

### Common Error 5.1



#### Trying to Modify Arguments

The following method contains a common error: trying to modify an argument.

```

public static int addTax(double price, double rate)
{
 double tax = price * rate / 100;
 price = price + tax; // Has no effect outside the method
}

```

```

 return tax;
 }
}

```

Now consider this call:

```

double total = 10;
addTax(total, 7.5); // Does not modify total

```

When the `addTax` method is called, `price` is set to 10. Then `price` is changed to 10.75. When the method returns, all of its parameter variables are removed. Any values that have been assigned to them are simply forgotten. Note that `total` is *not* changed. In Java, a method can never change the contents of a variable that was passed as an argument.

## 5.4 Return Values

The `return` statement terminates a method call and yields the method result.

You use the `return` statement to specify the result of a method. In the preceding examples, each `return` statement returned a variable. However, the `return` statement can return the value of any expression. Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return a more complex expression:

```

public static double cubeVolume(double sideLength)
{
 return sideLength * sideLength * sideLength;
}

```

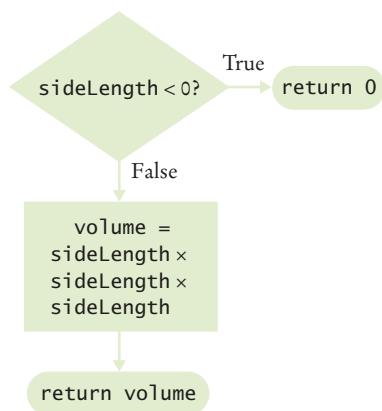
When the `return` statement is processed, the method exits *immediately*. Some programmers find this behavior convenient for handling exceptional cases at the beginning of the method:

```

public static double cubeVolume(double sideLength)
{
 if (sideLength < 0) { return 0; }
 // Handle the regular case
 .
}

```

If the method is called with a negative value for `sideLength`, then the method returns 0 and the remainder of the method is not executed. (See Figure 4.)



**Figure 4** A `return` Statement Exits a Method Immediately

Every branch of a method needs to return a value. Consider the following incorrect method:

```
public static double cubeVolume(double sideLength)
{
 if (sideLength >= 0)
 {
 return sideLength * sideLength * sideLength;
 } // Error—no return value if sideLength < 0
}
```

The compiler reports this as an error. A correct implementation is:

```
public static double cubeVolume(double sideLength)
{
 if (sideLength >= 0)
 {
 return sideLength * sideLength * sideLength;
 }
 else
 {
 return 0;
 }
}
```

Many programmers dislike the use of multiple return statements in a method. You can avoid multiple returns by storing the method result in a variable that you return in the last statement of the method. For example:

```
public static double cubeVolume(double sideLength)
{
 double volume;
 if (sideLength >= 0)
 {
 volume = sideLength * sideLength * sideLength;
 }
 else
 {
 volume = 0;
 }
 return volume;
}
```



**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program showing a method with multiple return statements.

### SELF CHECK



13. Suppose we change the body of the `cubeVolume` method to

```
if (sideLength <= 0) { return 0; }
return sideLength * sideLength * sideLength;
```

How does this method differ from the one described in this section?

14. What does this method do?

```
public static boolean mystery(int n)
{
 if (n % 2 == 0) { return true; }
 else { return false; }
}
```

15. Implement the `mystery` method of Self Check 14 with a single `return` statement.

### Practice It

Now you can try these exercises at the end of the chapter: R5.13, P5.8.

**Common Error 5.2****Missing Return Value**

It is a compile-time error if some branches of a method return a value and others do not. Consider this example:

```
public static int sign(double number)
{
 if (number < 0) { return -1; }
 if (number > 0) { return 1; }
 // Error: missing return value if number equals 0
}
```

This method computes the sign of a number:  $-1$  for negative numbers and  $+1$  for positive numbers. If the argument is zero, however, no value is returned. The remedy is to add a statement `return 0;` to the end of the method.

**HOW TO 5.1****Implementing a Method**

A method is a computation that can be used multiple times with different arguments, either in the same program or in different programs. Whenever a computation is needed more than once, turn it into a method.

**Problem Statement** Suppose that you are helping archaeologists who research Egyptian pyramids. You have taken on the task of writing a method that determines the volume of a pyramid, given its height and base length.



© Holger Mette/Stockphoto.

**Step 1** Describe what the method should do.

Provide a simple English description, such as “Compute the volume of a pyramid whose base is a square.”

**Step 2** Determine the method’s “inputs”.

Make a list of *all* the parameters that can vary. It is common for beginners to implement methods that are overly specific. For example, you may know that the great pyramid of Giza, the largest of the Egyptian pyramids, has a height of 146 meters and a base length of 230 meters. You should *not* use these numbers in your calculation, even if the original problem only asked about the great pyramid. It is just as easy—and far more useful—to write a method that computes the volume of *any* pyramid.

Turn computations  
that can be reused  
into methods.

In our case, the parameters are the pyramid’s height and base length. At this point, we have enough information to document the method:

```
/**
 * Computes the volume of a pyramid whose base is a square.
 * @param height the height of the pyramid
 * @param baseLength the length of one side of the pyramid's base
 * @return the volume of the pyramid
 */
```

**Step 3** Determine the types of the parameter variables and the return value.

The height and base length can both be floating-point numbers. Therefore, we will choose the type `double` for both parameter variables. The computed volume is also a floating-point number, yielding a return type of `double`. Therefore, the method will be declared as

```
public static double pyramidVolume(double height, double baseLength)
```

**Step 4** Write pseudocode for obtaining the desired result.

In most cases, a method needs to carry out several steps to find the desired answer. You may need to use mathematical formulas, branches, or loops. Express your method in pseudocode.

An Internet search yields the fact that the volume of a pyramid is computed as

$$\text{volume} = \frac{1}{3} \times \text{height} \times \text{base area}$$

Because the base is a square, we have

$$\text{base area} = \text{base length} \times \text{base length}$$

Using these two equations, we can compute the volume from the arguments.

**Step 5** Implement the method body.

In our example, the method body is quite simple. Note the use of the `return` statement to return the result.

```
public static double pyramidVolume(double height, double baseLength)
{
 double baseArea = baseLength * baseLength;
 return height * baseArea / 3;
}
```

**Step 6** Test your method.

After implementing a method, you should test it in isolation. Such a test is called a **unit test**. Work out test cases by hand, and make sure that the method produces the correct results. For example, for a pyramid with height 9 and base length 10, we expect the area to be  $\frac{1}{3} \times 9 \times 100 = 300$ . If the height is 0, we expect an area of 0.

```
public static void main(String[] args)
{
 System.out.println("Volume: " + pyramidVolume(9, 10));
 System.out.println("Expected: 300");
 System.out.println("Volume: " + pyramidVolume(0, 10));
 System.out.println("Expected: 0");
}
```

The output confirms that the method worked as expected:

```
Volume: 300
Expected: 300
Volume: 0
Expected: 0
```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download the program for calculating a pyramid's volume.

**WORKED EXAMPLE 5.1****Generating Random Passwords**

This Worked Example creates a method that generates passwords of a given length with at least one digit and one special character. Go to [wiley.com/go/bj102examples](http://wiley.com/go/bj102examples) and download Worked Example 5.1.

|                              |                      |
|------------------------------|----------------------|
| Enter your current password: | <input type="text"/> |
| Enter your new password:     | <input type="text"/> |
| Retype your new password:    | <input type="text"/> |

## 5.5 Methods Without Return Values

Use a return type of `void` to indicate that a method does not return a value.

Sometimes, you need to carry out a sequence of instructions that does not yield a value. If that instruction sequence occurs multiple times, you will want to package it into a method. In Java, you use the return type `void` to indicate the absence of a return value.

Here is a typical example: Your task is to print a string in a box, like this:

```

!Hello!

```



© jgroup/iStockphoto.

*A void method returns no value, but it can produce output.*

However, different strings can be substituted for `Hello`. A method for this task can be declared as follows:

```
public static void boxString(String contents)
```

Now you develop the body of the method in the usual way, by formulating a general method for solving the task.

**`n` = the length of the string**

**Print a line that contains the - character `n + 2` times.**

**Print a line containing the contents, surrounded with a ! to the left and right.**

**Print another line containing the - character `n + 2` times.**

Here is the method implementation:

```
/**
 * Prints a string in a box.
 * @param contents the string to enclose in a box
 */
public static void boxString(String contents)
{
 int n = contents.length();
 for (int i = 0; i < n + 2; i++) { System.out.print("-"); }
 System.out.println();
 System.out.println("!" + contents + "!");
 for (int i = 0; i < n + 2; i++) { System.out.print("-"); }
 System.out.println();
}
```

Note that this method doesn't compute any value. It performs some actions and then returns to the caller.

Because there is no return value, you cannot use `boxString` in an expression. You can call

```
boxString("Hello");
```

but not

```
result = boxString("Hello"); // Error: boxString doesn't return a result.
```

If you want to return from a void method before reaching the end, you use a `return` statement without a value. For example,



### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a complete program demonstrating the `boxString` method.

```
public static void boxString(String contents)
{
 int n = contents.length();
 if (n == 0)
 {
 return; // Return immediately
 }
 ...
}
```

**SELF CHECK**

- 16.** How do you generate the following printout, using the `boxString` method?

```

!Hello!

!World!

```

- 17.** What is wrong with the following statement?

```
System.out.print(boxString("Hello"));
```

- 18.** Implement a method `shout` that prints a line consisting of a string followed by three exclamation marks. For example, `shout("Hello")` should print `Hello!!!`. The method should not return a value.

- 19.** How would you modify the `boxString` method to leave a space around the string that is being boxed, like this:

```

! Hello !


```

- 20.** The `boxString` method contains the code for printing a line of - characters twice. Place that code into a separate method `printLine`, and use that method to simplify `boxString`. What is the code of both methods?

**Practice It** Now you can try these exercises at the end of the chapter: R5.4, P5.22.

## 5.6 Problem Solving: Reusable Methods

Eliminate replicated code or pseudocode by defining a method.

You have used many methods from the standard Java library. These methods have been provided as a part of the Java platform so that programmers need not recreate them. Of course, the Java library doesn't cover every conceivable need. You will often be able to save yourself time by designing your own methods that can be used for multiple problems.

When you write nearly identical code or pseudocode multiple times, either in the same program or in separate programs, consider introducing a method. Here is a typical example of code replication:

```
int hours;
do
{
 System.out.print("Enter a value between 0 and 23: ");
 hours = in.nextInt();
}
while (hours < 0 || hours > 23);
```

```

int minutes;
do
{
 System.out.print("Enter a value between 0 and 59: ");
 minutes = in.nextInt();
}
while (minutes < 0 || minutes > 59);

```

This program segment reads two variables, making sure that each of them is within a certain range. It is easy to extract the common behavior into a method:

```

/**
 * Prompts a user to enter a value up to a given maximum until the user
 * provides a valid input.
 * @param high the largest allowable input
 * @return the value provided by the user (between 0 and high, inclusive)
 */
public static int readIntUpTo(int high)
{
 int input;
 Scanner in = new Scanner(System.in);
 do
 {
 System.out.print("Enter a value between 0 and " + high + ": ");
 input = in.nextInt();
 }
 while (input < 0 || input > high);
 return input;
}

```

Then use this method twice:

```

int hours = readIntUpTo(23);
int minutes = readIntUpTo(59);

```

We have now removed the replication of the loop—it only occurs once, inside the method.

Note that the method can be reused in other programs that need to read integer values. However, we should consider the possibility that the smallest value need not always be zero.

Here is a better alternative:

```

/**
 * Prompts a user to enter a value within a given range until the user
 * provides a valid input.
 * @param low the smallest allowable input
 * @param high the largest allowable input
 * @return the value provided by the user (between low and high, inclusive)
 */
public static int readIntBetween(int low, int high)
{
 int input;
 Scanner in = new Scanner(System.in);
 do
 {
 System.out.print("Enter a value between " + low + " and " + high + ": ");
 input = in.nextInt();
 }
 while (input < low || input > high);
 return input;
}

```

**Design your methods to be reusable.**  
Supply parameter variables for the values that can vary when the method is reused.

*When carrying out the same task multiple times, use a method.*



© Lawrence Sawyer/Stockphoto.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a complete program that demonstrates the `readIntBetween` method.

#### SELF CHECK



In our program, we call

```
int hours = readIntBetween(0, 23);
```

Another program can call

```
int month = readIntBetween(1, 12);
```

In general, you will want to provide parameter variables for the values that vary when a method is reused.

- 21.** Consider the following statements:

```
int totalPennies = (int) Math.round(100 * total) % 100;
int taxPennies = (int) Math.round(100 * (total * taxRate)) % 100;
```

Introduce a method to reduce code duplication.

- 22.** Consider this method that prints a page number on the left or right side of a page:

```
if (page % 2 == 0) { System.out.println(page); }
else { System.out.println(" " + page); }
```

Introduce a method with return type boolean to make the condition in the if statement easier to understand.

- 23.** Consider the following method that computes compound interest for an account with an initial balance of \$10,000 and an interest rate of 5 percent:

```
public static double balance(int years) { return 10000 * Math.pow(1.05, years); }
```

How can you make this method more reusable?

- 24.** The comment explains what the following loop does. Use a method instead.

```
// Counts the number of spaces
int spaces = 0;
for (int i = 0; i < input.length(); i++)
{
 if (input.charAt(i) == ' ') { spaces++; }
```

- 25.** In Self Check 24, you were asked to implement a method that counts spaces. How can you generalize it so that it can count any character? Why would you want to do this?

#### Practice It

Now you can try these exercises at the end of the chapter: R5.7, P5.9.



## Computing & Society 5.1 Personal Computing

In 1971, Marcian E. "Ted" Hoff, an engineer at Intel Corporation, was working on a chip for a manufacturer of electronic calculators. He realized that it would be a better idea to develop a *general-purpose* chip that could be *programmed* to interface with the keys and display of a calculator, rather than to do yet another custom design. Thus, the *microprocessor* was born. At the time, its primary application was as a controller for calculators, washing machines, and the like. It took years for the computer industry to notice that a genuine central processing unit was now available as a single chip.

Hobbyists were the first to catch on. In 1974 the first computer *kit*, the Altair 8800, was available from MITS Electronics for about \$350. The kit consisted of the microprocessor, a circuit board, a very small amount of memory, toggle switches, and a row of

display lights. Purchasers had to solder and assemble it, then program it in machine language through the toggle switches. It was not a big hit.

The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a \$3,000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not even support lowercase letters, making it worthless for word processing. The breakthrough came in 1979 with a new spreadsheet program, VisiCalc. In a spreadsheet, you enter financial data and their relationships into a grid of rows and columns (see the figure). Then you modify some of the data and watch in real time how the others change. For example, you can see how changing the mix of widgets in a manufacturing plant might affect estimated

costs and profits. Corporate managers snapped up VisiCalc and the computer that was needed to run it. For them, the computer was a spreadsheet machine.

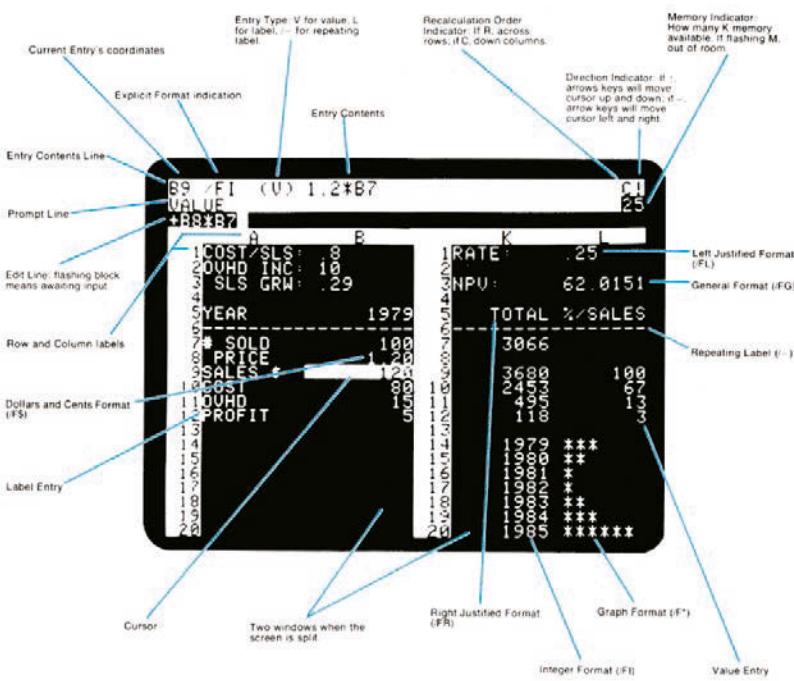
More importantly, it was a personal device. The managers were free to do the calculations that they wanted to do, not just the ones that the "high priests" in the data center provided.

Personal computers have been with us ever since, and countless users have tinkered with their hardware and software, sometimes establishing highly successful companies or creating free software for millions of users. This "freedom to tinker" is an important part of personal computing. On a personal device, you should be able to install the software that you want to install to make you more productive or creative, even if that's not the same software that most people use. You should be able to add peripheral equipment of your choice. For the first thirty years of personal computing, this freedom was largely taken for granted.

We are now entering an era where smart phones, tablets, and smart TV sets are replacing functions that were traditionally fulfilled by personal computers. While it is amazing to carry more computing power in your cell phone than in the best personal computers of the 1990s, it is disturbing that we lose a degree of personal control. With some phone or tablet brands, you can install only those applications that the manufacturer publishes on the "app store". For example, Apple rejected MIT's iPad app for the educational language Scratch because it contained a virtual machine. You'd think it would be in Apple's interest to encourage the next generation to be enthusiastic about programming, but they have a general policy of denying programmability on "their" devices, in order to thwart competitive environments such as Flash or Java.

When you select a device for making phone calls or watching movies, it is worth asking who is in control. Are you purchasing a personal device that you can use in any way you choose, or are you being tethered to a flow of data that is controlled by somebody else?

### A VISICALC™ Screen:



The VisiCalc Spreadsheet Running on an Apple II

## 5.7 Problem Solving: Stepwise Refinement

Use the process of stepwise refinement to decompose complex tasks into simpler ones.

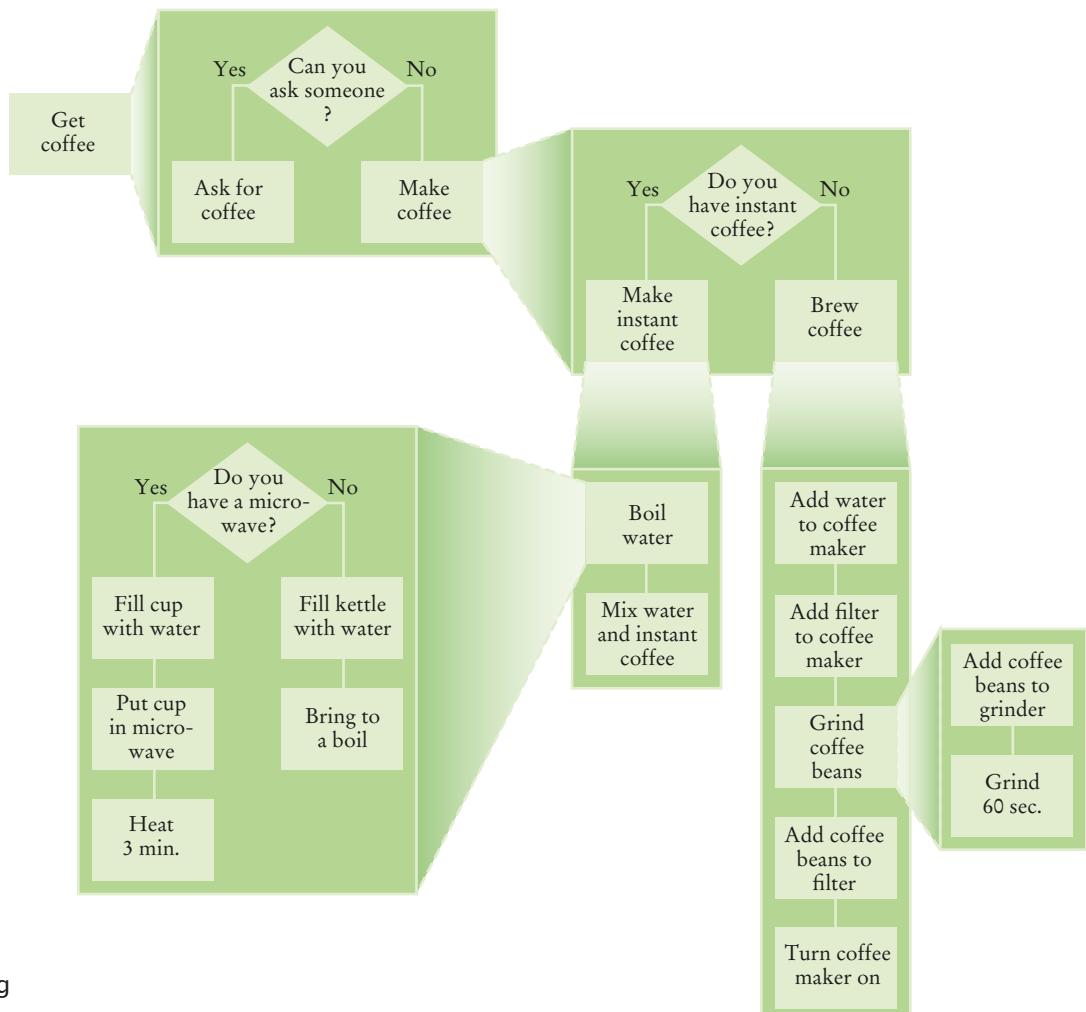
One of the most powerful strategies for problem solving is the process of **stepwise refinement**. To solve a difficult task, break it down into simpler tasks. Then keep breaking down the simpler tasks into even simpler ones, until you are left with tasks that you know how to solve.

Now apply this process to a problem of everyday life. You get up in the morning and simply must **get coffee**. How do you get coffee? You see whether you can get someone else, such as your mother or mate, to bring you some. If that fails, you must **make coffee**. How do you make coffee? If there is instant



© AdShooter/Stockphoto.

A production process is broken down into sequences of assembly steps.



**Figure 5**  
Flowchart of  
Coffee-Making  
Solution

coffee available, you can **make instant coffee**. How do you make instant coffee? Simply **boil water** and mix the boiling water with the instant coffee. How do you boil water? If there is a microwave, then you fill a cup with water, place it in the microwave and heat it for three minutes. Otherwise, you fill a kettle with water and heat it on the stove until the water comes to a boil. On the other hand, if you don't have instant coffee, you must **brew coffee**. How do you brew coffee? You add water to the coffee maker, put in a filter, **grind coffee**, put the coffee in the filter, and turn the coffee maker on. How do you grind coffee? You add coffee beans to the coffee grinder and push the button for 60 seconds.

Figure 5 shows a flowchart view of the coffee-making solution. Refinements are shown as expanding boxes. In Java, you implement a refinement as a method. For example, a method `brewCoffee` would call `grindCoffee`, and `brewCoffee` would be called from a method `makeCoffee`.

Let us apply the process of stepwise refinement to a programming problem. When printing a check, it is customary to write the check amount both as a number ("274.15") and as a text string ("two hundred seventy four dollars and 15 cents"). Doing so reduces the recipient's temptation to add a few digits in front of the amount.

For a human, this isn't particularly difficult, but how can a computer do this? There is no built-in method that turns 274 into "two hundred seventy four". We need to program this method. Here is the description of the method we want to write:

```
/**
 * Turns a number into its English name.
 * @param number a positive integer < 1,000
 * @return the name of number (e.g., "two hundred seventy four")
 */
public static String intName(int number)
```

When you discover that you need a method, write a description of the parameter variables and return values.

How can this method do its job? Consider a simple case first. If the number is between 1 and 9, we need to compute "one" ... "nine". In fact, we need the same computation *again* for the hundreds (two hundred). Any time you need something more than once, it is a good idea to turn that into a method. Rather than writing the entire method, write only the comment:

```
/**
 * Turns a digit into its English name.
 * @param digit an integer between 1 and 9
 * @return the name of digit ("one" ... "nine")
 */
public static String digitName(int digit)
```

A method may require simpler methods to carry out its work.

Numbers between 10 and 19 are special cases. Let's have a separate method `teenName` that converts them into strings "eleven", "twelve", "thirteen", and so on:

```
/**
 * Turns a number between 10 and 19 into its English name.
 * @param number an integer between 10 and 19
 * @return the name of the number ("ten" ... "nineteen")
 */
public static String teenName(int number)
```



© Yin Yang/iStockphoto.

Next, suppose that the number is between 20 and 99. The name of such a number has two parts, such as "seventy four". We need a way of producing the first part, "twenty", "thirty", and so on. Again, we will put that computation into a separate method:

```
/***
 Gives the name of the tens part of a number between 20 and 99.
 @param number an integer between 20 and 99
 @return the name of the tens part of the number ("twenty" ... "ninety")
*/
public static String tensName(int number)
```

Now let us write the pseudocode for the `intName` method. If the number is between 100 and 999, then we show a digit and the word "hundred" (such as "two hundred"). We then remove the hundreds, for example reducing 274 to 74. Next, suppose the remaining part is at least 20 and at most 99. If the number is evenly divisible by 10, we use `tensName`, and we are done. Otherwise, we print the tens with `tensName` (such as "seventy") and remove the tens, reducing 74 to 4. In a separate branch, we deal with numbers that are at between 10 and 19. Finally, we print any remaining single digit (such as "four").

```
intName(number)
part = number //The part that still needs to be converted
name = ""

If part >= 100
 name = name of hundreds in part + " hundred"
 Remove hundreds from part.

If part >= 20
 Append tensName(part) to name.
 Remove tens from part.
Else if part >= 10
 Append teenName(part) to name.
 part = 0

If (part > 0)
 Append digitName(part) to name.
```

Translating the pseudocode into Java is straightforward. The result is shown in the source listing at the end of this section.

Note how we rely on helper methods to do much of the detail work. Using the process of stepwise refinement, we now need to consider these helper methods.

Let's start with the `digitName` method. This method is so simple to implement that pseudocode is not really required. Simply use an `if` statement with nine branches:

```
public static String digitName(int digit)
{
 if (digit == 1) { return "one" };
 if (digit == 2) { return "two" };
 ...
}
```

The `teenName` and `tensName` methods are similar.

This concludes the process of stepwise refinement. Here is the complete program:

### sec07/IntegerName.java

```

1 import java.util.Scanner;
2
3 /**
4 * This program turns an integer into its English name.
5 */
6 public class IntegerName
7 {
8 public static void main(String[] args)
9 {
10 Scanner in = new Scanner(System.in);
11 System.out.print("Please enter a positive integer < 1000: ");
12 int input = in.nextInt();
13 System.out.println(intName(input));
14 }
15
16 /**
17 * Turns a number into its English name.
18 * @param number a positive integer < 1,000
19 * @return the name of the number (e.g. "two hundred seventy four")
20 */
21 public static String intName(int number)
22 {
23 int part = number; // The part that still needs to be converted
24 String name = ""; // The name of the number
25
26 if (part >= 100)
27 {
28 name = digitName(part / 100) + " hundred";
29 part = part % 100;
30 }
31
32 if (part >= 20)
33 {
34 name = name + " " + tensName(part);
35 part = part % 10;
36 }
37 else if (part >= 10)
38 {
39 name = name + " " + teenName(part);
40 part = 0;
41 }
42
43 if (part > 0)
44 {
45 name = name + " " + digitName(part);
46 }
47
48 return name;
49 }
50
51 /**
52 * Turns a digit into its English name.
53 * @param digit an integer between 1 and 9
54 * @return the name of digit ("one" ... "nine")
55 */

```

```

56 public static String digitName(int digit)
57 {
58 if (digit == 1) { return "one"; }
59 if (digit == 2) { return "two"; }
60 if (digit == 3) { return "three"; }
61 if (digit == 4) { return "four"; }
62 if (digit == 5) { return "five"; }
63 if (digit == 6) { return "six"; }
64 if (digit == 7) { return "seven"; }
65 if (digit == 8) { return "eight"; }
66 if (digit == 9) { return "nine"; }
67 return "";
68 }
69
70 /**
71 * Turns a number between 10 and 19 into its English name.
72 * @param number an integer between 10 and 19
73 * @return the name of the given number ("ten" ... "nineteen")
74 */
75 public static String teenName(int number)
76 {
77 if (number == 10) { return "ten"; }
78 if (number == 11) { return "eleven"; }
79 if (number == 12) { return "twelve"; }
80 if (number == 13) { return "thirteen"; }
81 if (number == 14) { return "fourteen"; }
82 if (number == 15) { return "fifteen"; }
83 if (number == 16) { return "sixteen"; }
84 if (number == 17) { return "seventeen"; }
85 if (number == 18) { return "eighteen"; }
86 if (number == 19) { return "nineteen"; }
87 return "";
88 }
89
90 /**
91 * Gives the name of the tens part of a number between 20 and 99.
92 * @param number an integer between 20 and 99
93 * @return the name of the tens part of the number ("twenty" ... "ninety")
94 */
95 public static String tensName(int number)
96 {
97 if (number >= 90) { return "ninety"; }
98 if (number >= 80) { return "eighty"; }
99 if (number >= 70) { return "seventy"; }
100 if (number >= 60) { return "sixty"; }
101 if (number >= 50) { return "fifty"; }
102 if (number >= 40) { return "forty"; }
103 if (number >= 30) { return "thirty"; }
104 if (number >= 20) { return "twenty"; }
105 return "";
106 }
107 }
```

### Program Run

Please enter a positive integer < 1000: 729  
seven hundred twenty nine



- SELF CHECK**
26. Explain how you can improve the `intName` method so that it can handle arguments up to 9999.
  27. Why does line 40 set `part = 0`?
  28. What happens when you call `intName(0)`? How can you change the `intName` method to handle this case correctly?
  29. Trace the method call `intName(72)`, as described in Programming Tip 5.4.
  30. Use the process of stepwise refinement to break down the task of printing the following table into simpler tasks.

| i  | $i * i * i$ |
|----|-------------|
| 1  | 1           |
| 2  | 8           |
| .  | .           |
| 20 | 8000        |

**Practice It** Now you can try these exercises at the end of the chapter: R5.12, P5.3, P5.21.

### Programming Tip 5.3



#### Keep Methods Short

There is a certain cost for writing a method. You need to design, code, and test the method. The method needs to be documented. You need to spend some effort to make the method reusable rather than tied to a specific context. To avoid this cost, it is always tempting just to stuff more and more code in one place rather than going through the trouble of breaking up the code into separate methods. It is quite common to see inexperienced programmers produce methods that are several hundred lines long.

As a rule of thumb, a method that is so long that its code will not fit on a single screen in your development environment should probably be broken up.

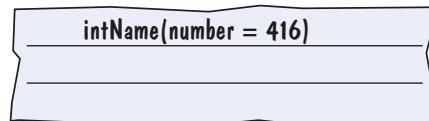
### Programming Tip 5.4



#### Tracing Methods

When you design a complex method, it is a good idea to carry out a manual walkthrough before entrusting your program to the computer.

Take an index card, or some other piece of paper, and write down the method call that you want to study. Write the name of the method and the names and values of the parameter variables, like this:



Then write the names and initial values of the method variables. Write them in a table, because you will update them as you walk through the code.

| intName(number = 416) |      |
|-----------------------|------|
| part                  | name |
| 416                   | ""   |
|                       |      |

We enter the test part  $\geq 100$ . part / 100 is 4 and part % 100 is 16. digitName(4) is easily seen to be "four". (Had digitName been complicated, you would have started another sheet of paper to figure out that method call. It is quite common to accumulate several sheets in this way.)

Now name has changed to name + " " + digitName(part / 100) + " hundred", that is "four hundred", and part has changed to part % 100, or 16.

| intName(number = 416) |                |
|-----------------------|----------------|
| part                  | name           |
| 416                   | " "            |
| 16                    | "four hundred" |
|                       |                |
|                       |                |

Now you enter the branch part  $\geq 10$ . teenName(16) is sixteen, so the variables now have the values

| intName(number = 416) |                        |
|-----------------------|------------------------|
| part                  | name                   |
| 416                   | " "                    |
| 16                    | "four hundred"         |
| 0                     | "four hundred sixteen" |
|                       |                        |

Now it becomes clear why you need to set part to 0 in line 40. Otherwise, you would enter the next branch and the result would be "four hundred sixteen six". Tracing the code is an effective way to understand the subtle aspects of a method.

### Programming Tip 5.5



#### Stubs

When writing a larger program, it is not always feasible to implement and test all methods at once. You often need to test a method that calls another, but the other method hasn't yet been implemented. Then you can temporarily replace the missing method with a **stub**. A stub is a method that returns a simple value that is sufficient for testing another method. Here are examples of stub methods:

```
/**
 * Turns a digit into its English name.
 * @param digit an integer between 1 and 9
 * @return the name of digit ("one" ... "nine")
 */
public static String digitName(int digit)
{
 return "mumble";
}

/**
 * Gives the name of the tens part of a number between 20 and 99.
 * @param number an integer between 20 and 99
 * @return the tens name of the number ("twenty" ... "ninety")
```



Stubs are incomplete methods that can be used for testing.

© iStockphotography/Stockphoto.

```

 */
public static String tensName(int number)
{
 return "mumblety";
}

```

If you combine these stubs with the `intName` method and test it with an argument of 274, you will get a result of "mumble hundred mumblety mumble", which indicates that the basic logic of the `intName` method is working correctly.



### WORKED EXAMPLE 5.2

### Calculating a Course Grade



Learn how to use stepwise refinement to solve the problem of converting a set of letter grades into an average grade for a course. Go to [wiley.com/go/bj102examples](http://wiley.com/go/bj102examples) and download Worked Example 5.2.



© Paul Kline/  
iStockphoto.

## 5.8 Variable Scope

The scope of a variable is the part of the program in which it is visible.

As your programs get larger and contain more variables, you may encounter problems where you cannot access a variable that is defined in a different part of your program, or where two variable definitions conflict with each other. In order to resolve these problems, you need to be familiar with the concept of *variable scope*.

The **scope** of a variable is the part of the program in which you can access it. For example, the scope of a method's parameter variable is the entire method. In the following code segment, the scope of the parameter variable `sideLength` is the entire `cubeVolume` method but not the `main` method.

```

public static void main(String[] args)
{
 System.out.println(cubeVolume(10));
}

public static double cubeVolume(double sideLength)
{
 return sideLength * sideLength * sideLength;
}

```

A variable that is defined within a method is called a **local variable**. When a local variable is declared in a block, its scope ranges from its declaration until the end of the block. For example, in the code segment below, the scope of the `square` variable is highlighted.

```

public static void main(String[] args)
{
 int sum = 0;
 for (int i = 1; i <= 10; i++)
 {
 int square = i * i;
 sum = sum + square;
 }
 System.out.println(sum);
}

```

The scope of a variable declared in a `for` statement extends to the end of the statement:

```
public static void main(String[] args)
{
 int sum = 0;
 for (int i = 1; i <= 10; i++)
 {
 sum = sum + i * i;
 }
 System.out.println(sum);
}
```

Here is an example of a scope problem. The following code will not compile:

```
public static void main(String[] args)
{
 double sideLength = 10;
 int result = cubeVolume();
 System.out.println(result);
}

public static double cubeVolume()
{
 return sideLength * sideLength * sideLength; // ERROR
}
```

Note the scope of the variable `sideLength`. The `cubeVolume` method attempts to read the variable, but it cannot—the scope of `sideLength` does not extend outside the `main` method. The remedy is to pass it as an argument, as we did in Section 5.2.

It is possible to use the same variable name more than once in a program. Consider the `result` variables in the following example:

```
public static void main(String[] args)
{
 int result = square(3) + square(4);
 System.out.println(result);
}

public static int square(int n)
{
 int result = n * n;
 return result;
}
```

Each `result` variable is declared in a separate method, and their scopes do not overlap.

© jcchamp/iStockphoto (Railway and Main);  
 © StevenCarrieJohansson/iStockphoto (Main and N. Putnam);  
 © jsmith/iStockphoto (Main and South St.)



*In the same way that there can be a street named “Main Street” in different cities, a Java program can have multiple variables with the same name.*

Two local or parameter variables can have the same name, provided that their scopes do not overlap.

You can even have two variables with the same name in the same method, provided that their scopes do not overlap:

```
public static void main(String[] args)
{
 int sum = 0;
 for (int i = 1; i <= 10; i++)
 {
 sum = sum + i;
 }

 for (int i = 1; i <= 10; i++)
 {
 sum = sum + i * i;
 }
 System.out.println(sum);
}
```

It is not legal to declare two variables with the same name in the same method in such a way that their scopes overlap. For example, the following is not legal:

```
public static int sumOfSquares(int n)
{
 int sum = 0;
 for (int i = 1; i <= n; i++)
 {
 int n = i * i; // ERROR
 sum = sum + n;
 }
 return sum;
}
```

The scope of the local variable `n` is contained within the scope of the parameter variable `n`. In this case, you need to rename one of the variables.

### SELF CHECK



Consider this sample program, then answer the questions below.

```
1 public class Sample
2 {
3 public static void main(String[] args)
4 {
5 int x = 4;
6 x = mystery(x + 1);
7 System.out.println(s);
8 }
9
10 public static int mystery(int x)
11 {
12 int s = 0;
13 for (int i = 0; i < x; x++)
14 {
15 int x = i + 1;
16 s = s + x;
17 }
18 return s;
19 }
20 }
```

31. Which lines are in the scope of the variable `i` declared in line 13?
32. Which lines are in the scope of the parameter variable `x` declared in line 10?
33. The program declares two local variables with the same name whose scopes don't overlap. What are they?
34. There is a scope error in the `mystery` method. How do you fix it?
35. There is a scope error in the `main` method. What is it, and how do you fix it?

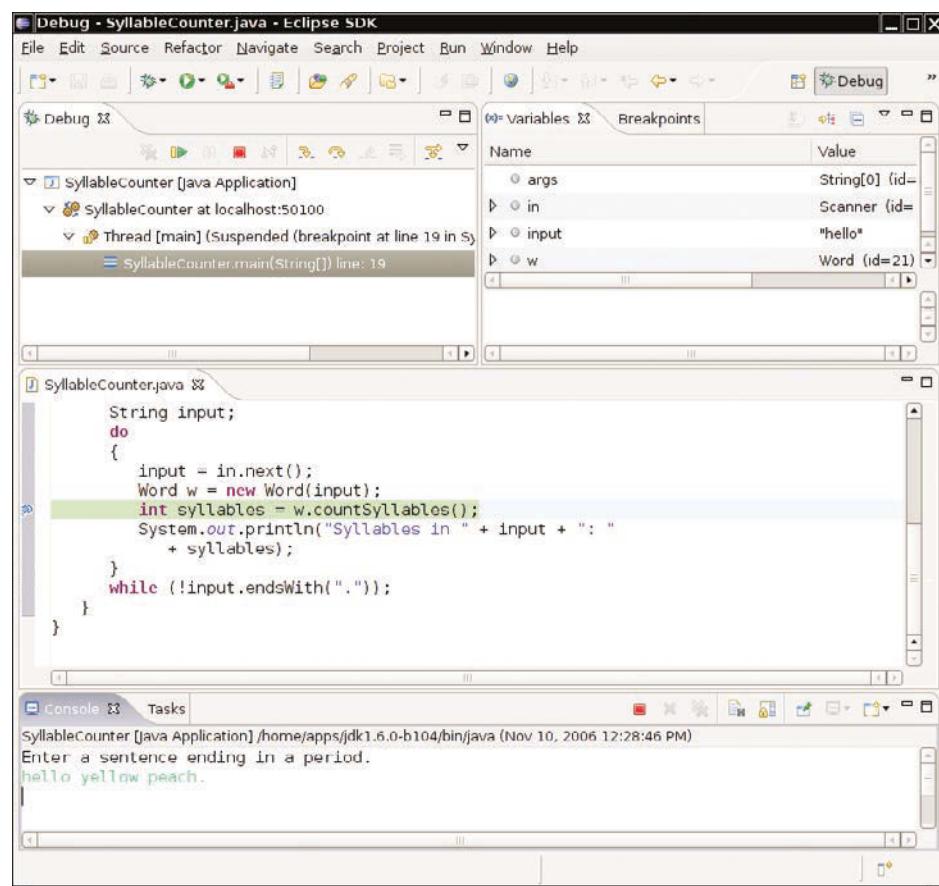
**Practice It**

Now you can try these exercises at the end of the chapter: R5.9, R5.10.

**Programming Tip 5.6****Using a Debugger**

As you have undoubtedly realized by now, computer programs rarely run perfectly the first time. At times, it can be quite frustrating to find the bugs. Of course, you can insert print commands, run the program, and try to analyze the printout. If the printout does not clearly point to the problem, you may need to add and remove print commands and run the program again. That can be a time-consuming process.

Modern development environments contain special programs, called **debuggers**, that help you locate bugs by letting you follow the execution of a program. You can stop your program and see the contents of variables, then run the program for more steps and observe how the variables change. Figure 6 shows the debugger from the Eclipse development environment.



**Figure 6** Stopping at a Breakpoint in Eclipse

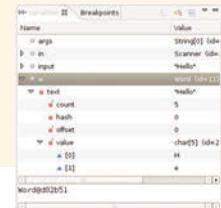
Debuggers vary somewhat from one development environment to another. You can go a long way with just a few commands that are typically called something like “Set breakpoint”, “Step to the next line”, “Step into the next method”, and “Run to the next breakpoint”. It is a good idea to learn how to use the debugger in your development environment. Then you can put it to use when you face a program that stubbornly refuses to do what it is supposed to. Video Example 5.1 walks you through a sample debugging session.



## VIDEO EXAMPLE 5.1

## Debugging

In this Video Example, you will learn how to use a debugger to find errors in a program. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 5.1.



## 5.9 Recursive Methods (Optional)

A recursive computation solves a problem by using the solution of the same problem with simpler inputs.

A recursive method is a method that calls itself. This is not as unusual as it sounds at first. Suppose you face the arduous task of cleaning up an entire house. You may well say to yourself, “I’ll pick a room and clean it, and then I’ll clean the other rooms.” In other words, the cleanup task calls itself, but with a simpler input. Eventually, all the rooms will be cleaned.

In Java, a recursive method uses the same principle. Here is a typical example. We want to print triangle patterns like this:

```
[]
[] []
[] [] []
[] [] [] []
```

Specifically, our task is to provide a method

```
public static void printTriangle(int sideLength)
```

The triangle given above is printed by calling `printTriangle(4)`. To see how recursion helps, consider how a triangle with side length 4 can be obtained from a triangle with side length 3.

```
[]
[] []
[] [] []
[] [] [] []
```

**Print the triangle with side length 3.**  
**Print a line with four [].**

More generally, here are the Java instructions for an arbitrary side length:

```
public static void printTriangle(int sideLength)
{
 printTriangle(sideLength - 1);
 for (int i = 0; i < sideLength; i++)
 {
```



© Janice Richard/iStockphoto.

```

 System.out.print("[]");
 }
 System.out.println();
}
}

```

There is just one problem with this idea. When the side length is 1, we don't want to call `printTriangle(0)`, `printTriangle(-1)`, and so on. The solution is simply to treat this as a special case, and not to print anything when `sideLength` is less than 1.

```

public static void printTriangle(int sideLength)
{
 if (sideLength < 1) { return; }
 printTriangle(sideLength - 1);
 for (int i = 0; i < sideLength; i++)
 {
 System.out.print("[]");
 }
 System.out.println();
}
}

```

Look at the `printTriangle` method one more time and notice how utterly reasonable it is. If the side length is 0, nothing needs to be printed. The next part is just as reasonable. Print the smaller triangle *and don't think about why that works*. Then print a row of `[]`. Clearly, the result is a triangle of the desired size.

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly.

The `printTriangle` method calls itself again with smaller and smaller side lengths. Eventually the side length must reach 0, and the method stops calling itself.

Here is what happens when we print a triangle with side length 4:

- The call `printTriangle(4)` calls `printTriangle(3)`.
- The call `printTriangle(3)` calls `printTriangle(2)`.
  - The call `printTriangle(2)` calls `printTriangle(1)`.
    - The call `printTriangle(1)` calls `printTriangle(0)`.
      - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
    - The call `printTriangle(3)` prints `[][][]`.
  - The call `printTriangle(4)` prints `[][][]`.

The call pattern of a recursive method looks complicated, and the key to the successful design of a recursive method is *not to think about it*.

For a recursion to terminate, there must be special cases for the simplest inputs.



© Niclae Popovici/  
iStockphoto

*This set of Russian dolls looks similar to the call pattern of a recursive method.*

Recursion is not really necessary to print triangle shapes. You can use nested loops, like this:

```
public static void printTriangle(int sideLength)
{
 for (int i = 0; i < sideLength; i++)
 {
 for (int j = 0; j < i; j++)
 {
 System.out.print("[]");
 }
 System.out.println();
 }
}
```

However, this pair of loops is a bit tricky. Many people find the recursive solution simpler to understand.

**SELF CHECK**


- 36.** Consider this slight modification of the `printTriangle` method:

```
public static void printTriangle(int sideLength)
{
 if (sideLength < 1) { return; }
 for (int i = 0; i < sideLength; i++)
 {
 System.out.print("[]");
 }
 System.out.println();
 printTriangle(sideLength - 1);
}
```

What is the result of `printTriangle(4)`?

- 37.** Consider this recursive method:

```
public static int mystery(int n)
{
 if (n <= 0) { return 0; }
 return n + mystery(n - 1);
}
```

What is `mystery(4)`?

- 38.** Consider this recursive method:

```
public static int mystery(int n)
{
 if (n <= 0) { return 0; }
 return mystery(n / 2) + 1;
}
```

What is `mystery(20)`?

- 39.** Write a recursive method for printing `n` box shapes `[]` in a row.

- 40.** The `intName` method in Section 5.7 accepted arguments  $< 1,000$ . Using a recursive call, extend its range to 999,999. For example an input of 12,345 should return "twelve thousand three hundred forty five".

**Practice It** Now you can try these exercises at the end of the chapter: R5.18, E5.11, E5.13.

## HOW TO 5.2



## Thinking Recursively

To solve a problem recursively requires a different mindset than to solve it by programming loops. In fact, it helps if you are, or pretend to be, a bit lazy and let others do most of the work for you. If you need to solve a complex problem, pretend that “someone else” will do most of the heavy lifting and solve the problem for all simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem.

**Problem Statement** To illustrate the recursive thinking process, consider the problem of Section 4.2, computing the sum of the digits of a number. We want to design a method `digitSum` that computes the sum of the digits of an integer  $n$ .

For example,  $\text{digitSum}(1729) = 1 + 7 + 2 + 9 = 19$ .

**Step 1**

The key to finding a recursive solution is reducing the input to a simpler input for the same problem.

Break the input into parts that can themselves be inputs to the problem.

In your mind, focus on a particular input or set of inputs for the task that you want to solve, and think how you can simplify the inputs. Look for simplifications that can be solved by the same task, and whose solutions are related to the original task.

In the digit sum problem, consider how we can simplify an input such as  $n = 1729$ . Would it help to subtract 1? After all,  $\text{digitSum}(1729) = \text{digitSum}(1728) + 1$ . But consider  $n = 1000$ . There seems to be no obvious relationship between  $\text{digitSum}(1000)$  and  $\text{digitSum}(999)$ .

A much more promising idea is to remove the last digit, that is, to compute  $n / 10 = 172$ . The digit sum of 172 is directly related to the digit sum of 1729.

**Step 2**

When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

Combine solutions with simpler inputs into a solution of the original problem.

In your mind, consider the solutions for the simpler inputs that you have discovered in Step 1. Don’t worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

In the case of the digit sum task, ask yourself how you can obtain  $\text{digitSum}(1729)$  if you know  $\text{digitSum}(172)$ . You simply add the last digit (9) and you are done. How do you get the last digit? As the remainder  $n \% 10$ . The value  $\text{digitSum}(n)$  can therefore be obtained as

$$\text{digitSum}(n / 10) + n \% 10$$

Don’t worry how  $\text{digitSum}(n / 10)$  is computed. The input is smaller, and therefore it works.

**Step 3**

Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them. That is usually very easy.

Look at the simplest inputs for the `digitSum` problem:

- A number with a single digit
- 0

A number with a single digit is its own digit sum, so you can stop the recursion when  $n < 10$ , and return  $n$  in that case. Or, you can be even lazier. If  $n$  has a single digit, then  $\text{digitSum}(n / 10) + n \% 10$  equals  $\text{digitSum}(0) + n$ . You can simply terminate the recursion when  $n$  is zero.

**Step 4**

Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn’t one of the simplest cases, then implement the logic you discovered in Step 2.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program illustrating the digitSum method.

Here is the complete digitSum method:

```
public static int digitSum(int n)
{
 if (n == 0) { return 0; } // Special case for terminating the recursion
 return digitSum(n / 10) + n % 10; // General case
}
```

**VIDEO EXAMPLE 5.2****Fully Justified Text**

In printed books (such as this one), all but the last line of a paragraph have the same length. In this Video Example, you will see how to achieve this effect. Go to [wiley.com/go/bj102videos](http://wiley.com/go/bj102videos) to view Video Example 5.2.



© Kenneth C. Zirkel/  
iStockphoto.

**CHAPTER SUMMARY****Understand the concepts of methods, arguments, and return values.**

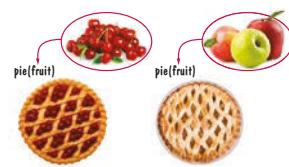
- A method is a named sequence of instructions.
- Arguments are supplied when a method is called.
- The return value is the result that the method computes.



- When declaring a method, you provide a name for the method, a variable for each argument, and a type for the result.
- Method comments explain the purpose of the method, the meaning of the parameter variables and return value, as well as any special requirements.

**Describe the process of parameter passing.**

- Parameter variables hold the arguments supplied in the method call.

**Describe the process of returning a value from a method.**

- The return statement terminates a method call and yields the method result.
- Turn computations that can be reused into methods.

**Design and implement methods without return values.**

- Use a return type of void to indicate that a method does not return a value.

**Develop methods that can be reused for multiple problems.**

- Eliminate replicated code or pseudocode by defining a method.
- Design your methods to be reusable. Supply parameter variables for the values that can vary when the method is reused.

### Apply the design principle of stepwise refinement.



- Use the process of stepwise refinement to decompose complex tasks into simpler ones.
- When you discover that you need a method, write a description of the parameter variables and return values.
- A method may require simpler methods to carry out its work.

### Determine the scope of variables in a program.

- The scope of a variable is the part of the program in which it is visible.
- Two local or parameter variables can have the same name, provided that their scopes do not overlap.



### Understand recursive method calls and implement simple recursive methods.



- A recursive computation solves a problem by using the solution of the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.
- The key to finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

## REVIEW EXERCISES

- **R5.1** In which sequence are the lines of the `Cubes.java` program in Section 5.2 executed, starting with the first line of `main`?
- **R5.2** Write method headers for methods with the following descriptions.
- Computing the larger of two integers
  - Computing the smallest of three floating-point numbers
  - Checking whether an integer is a prime number, returning `true` if it is and `false` otherwise
  - Checking whether a string is contained inside another string
  - Computing the balance of an account with a given initial balance, an annual interest rate, and a number of years of earning interest
  - Printing the balance of an account with a given initial balance and an annual interest rate over a given number of years
  - Printing the calendar for a given month and year
  - Computing the weekday for a given day, month, and year (as a string such as "Monday")
  - Generating a random integer between 1 and  $n$

- R5.3 Give examples of the following methods from the Java library.
- A method with a `double` argument and a `double` return value
  - A method with two `double` arguments and a `double` return value
  - A method with a `String` argument and a `double` return value
  - A method with no arguments and a `double` return value

- R5.4 True or false?

- A method has exactly one `return` statement.
- A method has at least one `return` statement.
- A method has at most one `return` value.
- A method with return value `void` never has a `return` statement.
- When executing a `return` statement, the method exits immediately.
- A method with return value `void` must print a result.
- A method without parameter variables always returns the same value.

- R5.5 Consider these methods:

```
public static double f(double x) { return g(x) + Math.sqrt(h(x)); }
public static double g(double x) { return 4 * h(x); }
public static double h(double x) { return x * x + k(x) - 1; }
public static double k(double x) { return 2 * (x + 1); }
```

Without actually compiling and running a program, determine the results of the following method calls.

- `double x1 = f(2);`
- `double x2 = g(h(2));`
- `double x3 = k(g(2) + h(2));`
- `double x4 = f(0) + f(1) + f(2);`
- `double x5 = f(-1) + g(-1) + h(-1) + k(-1);`

- R5.6 What is the difference between an argument and a return value? How many arguments can a method call have? How many return values?

- R5.7 Design a method that prints a floating-point number as a currency value (with a \$ sign and two decimal digits).

- Indicate how the programs `ch02/sec03/Volume2.java` and `ch04/sec03/InvestmentTable.java` should change to use your method.
- What change is required if the programs should show a different currency, such as euro?

- Business R5.8 Write pseudocode for a method that translates a telephone number with letters in it (such as 1-800-FLOWERS) into the actual phone number. Use the standard letters on a phone pad.

- R5.9 Describe the scope error in the following program and explain how to fix it.

```
public class Conversation
{
 public static void main(String[] args)
 {
```



© stacey\_newman/Stockphoto.

```

Scanner in = new Scanner(System.in);
System.out.print("What is your first name? ");
String input = in.next();
System.out.println("Hello, " + input);
System.out.print("How old are you? ");
int input = in.nextInt();
input++;
System.out.println("Next year, you will be " + input);
}
}

```

- R5.10** For each of the variables in the following program, indicate the scope. Then determine what the program prints, without actually running the program.

```

1 public class Sample
2 {
3 public static void main(String[] args)
4 {
5 int i = 10;
6 int b = g(i);
7 System.out.println(b + i);
8 }
9
10 public static int f(int i)
11 {
12 int n = 0;
13 while (n * n <= i) { n++; }
14 return n - 1;
15 }
16
17 public static int g(int a)
18 {
19 int b = 0;
20 for (int n = 0; n < a; n++)
21 {
22 int i = f(n);
23 b = b + i;
24 }
25 return b;
26 }
27 }

```

- R5.11** Use the process of stepwise refinement to describe the process of making scrambled eggs. Discuss what you do if you do not find eggs in the refrigerator.

- R5.12** Perform a walkthrough of the `intName` method with the following arguments:

- |               |               |
|---------------|---------------|
| <b>a.</b> 5   | <b>e.</b> 324 |
| <b>b.</b> 12  | <b>f.</b> 0   |
| <b>c.</b> 21  | <b>g.</b> -2  |
| <b>d.</b> 301 |               |

- R5.13** Consider the following method:

```

public static int f(int a)
{
 if (a < 0) { return -1; }
 int n = a;
 while (n > 0)
 {

```

```

 if (n % 2 == 0) // n is even
 {
 n = n / 2;
 }
 else if (n == 1) { return 1; }
 else { n = 3 * n + 1; }
 }
 return 0;
}

```

Perform traces of the computations  $f(-1)$ ,  $f(0)$ ,  $f(1)$ ,  $f(2)$ ,  $f(10)$ , and  $f(100)$ .

- R5.14 Consider the following method that is intended to swap the values of two integers:

```

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

public static void main(String[] args)
{
 int x = 3;
 int y = 4;
 falseSwap(x, y);
 System.out.println(x + " " + y);
}

```

Why doesn't the `falseSwap` method swap the contents of `x` and `y`?

- R5.15 In Worked Example 4.1, it is easy enough to measure the width of a real pyramid. But how can you measure the height without climbing to the top? You need to measure the angle between the ground and the top from a point, and the distance from that point to the base. Develop a concrete plan and write pseudocode for a program that determines the volume of a pyramid from these inputs.
- R5.16 Suppose an ancient civilization had constructed circular pyramids. Write pseudocode for a program that determines the surface area from measurements that you can determine from the ground.
- R5.17 Give pseudocode for a recursive method for printing all substrings of a given string. For example, the substrings of the string "rum" are "rum" itself, "ru", "um", "r", "u", "m", and the empty string. You may assume that all letters of the string are different.
- R5.18 Give pseudocode for a recursive method that sorts all letters in a string. For example, the string "goodbye" would be sorted into "bdegooy".

## PRACTICE EXERCISES

- E5.1 Write the following methods and provide a program to test them.

- `double smallest(double x, double y, double z)`, returning the smallest of the arguments
- `double average(double x, double y, double z)`, returning the average of the arguments

**E5.2** Write the following methods and provide a program to test them.

- a. boolean allTheSame(double x, double y, double z), returning true if the arguments are all the same
- b. boolean allDifferent(double x, double y, double z), returning true if the arguments are all different
- c. boolean sorted(double x, double y, double z), returning true if the arguments are sorted, with the smallest one coming first

**E5.3** Write the following methods.

- a. int firstDigit(int n), returning the first digit of the argument
- b. int lastDigit(int n), returning the last digit of the argument
- c. int digits(int n), returning the number of digits of the argument

For example, `firstDigit(1729)` is 1, `lastDigit(1729)` is 9, and `digits(1729)` is 4. Provide a program that tests your methods.

**E5.4** Write a method

```
public static String middle(String str)
```

that returns a string containing the middle character in `str` if the length of `str` is odd, or the two middle characters if the length is even. For example, `middle("middle")` returns "dd".

**E5.5** Write a method

```
public static String repeat(String str, int n)
```

that returns the string `str` repeated `n` times. For example, `repeat("ho", 3)` returns "hohoho".

**E5.6** Write a method

```
public static int countVowels(String str)
```

that returns a count of all vowels in the string `str`. Vowels are the letters a, e, i, o, and u, and their uppercase variants. Use a helper method `isVowel` that checks whether a character is a vowel.

**E5.7** Write a method

```
public static int countWords(String str)
```

that returns a count of all words in the string `str`. Words are separated by spaces. For example, `countWords("Mary had a little lamb")` should return 5. Your method should work correctly if there are multiple spaces between words. Use helper methods to find the position of the next space following a given position, and the position of the next non-space character following a given position.

**E5.8** Write a method that returns the average length of all words in the string `str`. Use the same helper methods as in Exercise E5.7.**E5.9** Write methods

```
public static double sphereVolume(double r)
public static double sphereSurface(double r)
public static double cylinderVolume(double r, double h)
public static double cylinderSurface(double r, double h)
```

```
public static double coneVolume(double r, double h)
public static double coneSurface(double r, double h)
```

that compute the volume and surface area of a sphere with radius  $r$ , a cylinder with a circular base with radius  $r$  and height  $h$ , and a cone with a circular base with radius  $r$  and height  $h$ . Then write a program that prompts the user for the values of  $r$  and  $h$ , calls the six methods, and prints the results.

**•• E5.10** Write a recursive method

```
public static String reverse(String str)
```

that computes the reverse of a string. For example, `reverse("flow")` should return `"wolf"`. *Hint:* Reverse the substring starting at the second character, then add the first character at the end. For example, to reverse `"flow"`, first reverse `"low"` to `"wl"`, then add the `"f"` at the end.

**•• E5.11** Write a recursive method

```
public static boolean isPalindrome(String str)
```

that returns true if `str` is a palindrome, that is, a word that is the same when reversed. Examples of palindrome are “deed”, “rotor”, or “aibohphobia”. *Hint:* A word is a palindrome if the first and last letters match and the remainder is also a palindrome.

**•• E5.12** Use recursion to implement a method `public static boolean find(String str, String match)` that tests whether `match` is contained in `str`:

```
boolean b = find("Mississippi", "sip"); // Sets b to true
```

*Hint:* If `str` starts with `match`, then you are done. If not, consider the string that you obtain by removing the first character.

**• E5.13** Use recursion to determine the number of digits in an integer `n`. *Hint:* If `n` is  $< 10$ , it has one digit. Otherwise, it has one more digit than `n / 10`.

**•• Business E5.14** Write a method that computes the balance of a bank account with a given initial balance and interest rate, after a given number of years. Assume interest is compounded yearly.

**•• E5.15** Write a program that prints instructions to get coffee, asking the user for input whenever a decision needs to be made. Decompose each task into a method, for example:

```
public static void brewCoffee()
{
 System.out.println("Add water to the coffee maker.");
 System.out.println("Put a filter in the coffee maker.");
 grindCoffee();
 System.out.println("Put the coffee in the filter.");
 ...
}
```

**•• E5.16** Write a method that computes the Scrabble score of a word. Look up the letter values on the Internet. For example, with English letter values, the word `JAVA` is worth  $8 + 1 + 4 + 1 = 14$  points.

**•• E5.17** Write a method that tests whether a file name should come before or after another. File names are first sorted by their extension (the string after the last period) and then by the name part (the string that remains after removing the extension). For example, `before("report.doc", "notes.txt")` should return `true` and `before("report.txt",`

"notes.txt") should return false. Provide helper methods that return the extension and name part of a file name.

- E5.18** When comparing strings with `compareTo`, the comparison is not always satisfactory. For example, "file10".`compareTo`("file2") returns a negative value, indicating that "file10" should come before "file2", even though we would prefer it to come afterwards. Produce a `numCompare` method that, when comparing two strings that are identical except for a positive integer at the end, compares the integers. For example, `numCompare("file12", "file2")` should return 1, but `numCompare("file12", "file11")` and `numCompare("file2", "doc12")` should return -1. Use a helper method that returns the starting position of the number, or -1 if there is none.

## PROGRAMMING PROJECTS

- P5.1** It is a well-known phenomenon that most people are easily able to read a text whose words have two characters flipped, provided the first and last letter of each word are not changed. For example,

I dn'ot gvie a dman for a man taht can olny sepll a wrod one way. (Mrak Taiwn)

Write a method `String scramble(String word)` that constructs a scrambled version of a given word, randomly flipping two characters other than the first and last one. Then write a program that reads words and prints the scrambled words.

- P5.2** Write a method

```
public static double readDouble(String prompt)
```

that displays the prompt string, followed by a space, reads a floating-point number in, and returns it. Here is a typical usage:

```
salary = readDouble("Please enter your salary:");
percentageRaise = readDouble("What percentage raise would you like?");
```

- P5.3** Enhance the `intName` method so that it works correctly for values < 1,000,000,000.
- P5.4** Enhance the `intName` method so that it works correctly for negative values and zero. *Caution:* Make sure the improved method doesn't print 20 as "twenty zero".
- P5.5** For some values (for example, 20), the `intName` method returns a string with a leading space (" twenty"). Repair that blemish and ensure that spaces are inserted only when necessary. *Hint:* There are two ways to do this. Either ensure that leading spaces are never inserted, or remove leading spaces from the result before returning it.
- P5.6** Write a method `String getTimeName(int hours, int minutes)` that returns the English name for a point in time, such as "ten minutes past two", "half past three", "a quarter to four", or "five o'clock". Assume that hours is between 1 and 12.
- P5.7** Use recursion to compute  $a^n$ , where  $n$  is a positive integer. *Hint:* If  $n$  is 1, then  $a^n = a$ . If  $n$  is even, then  $a^n = (a^{n/2})^2$ . Otherwise,  $a^n = a \times a^{n-1}$ .
- P5.8** *Leap years.* Write a method

```
public static boolean isLeapYear(int year)
```

that tests whether a year is a leap year: that is, a year with 366 days. Exercise P3.14 describes how to test whether a year is a leap year. In this exercise, use multiple `if` statements and return statements to return the result as soon as you know it.



- P5.9** In Exercise P3.13 you were asked to write a program to convert a number to its representation in Roman numerals. At the time, you did not know how to eliminate duplicate code, and as a consequence the resulting program was rather long. Rewrite that program by implementing and using the following method:

```
public static String romanDigit(int n, String one, String five, String ten)
```

That method translates one digit, using the strings specified for the one, five, and ten values. You would call the method as follows:

```
romanOnes = romanDigit(n % 10, "I", "V", "X");
n = n / 10;
romanTens = romanDigit(n % 10, "X", "L", "C");
...

```



© Straitshooter/iStockphoto.

- P5.10** Implement the `numberToGrade` method of Worked Example 5.2 so that you use two sets of branches: one to determine whether the grade should be A, B, C, D, or F, and another to determine whether a + or - should be appended.
- P5.11** Write a program that reads two strings containing section numbers such as "1.13.2" and "1.2.4.1" and determines which one comes first. Provide appropriate helper methods.
- P5.12** Write a program that prompts the user for a regular English verb (such as play), the first, second, or third person, singular or plural, and present, past, or future tense. Provide these values to a method that yields the conjugated verb and prints it. For example, the input `play 3 singular present` should yield "he/she plays".
- P5.13** Write a program that reads words and arranges them in a paragraph so that all lines other than the last one are exactly forty characters long. Add spaces between words to make the last word extend to the margin. Distribute the spaces evenly. Use a helper method for that purpose. A typical output would be:

```
Four score and seven years ago our
fathers brought forth on this continent
a new nation, conceived in liberty, and
dedicated to the proposition that all
men are created equal.
```

- P5.14** Write a program that, given a month and year, prints a calendar, such as

|           |    |    |    |    |    |    |
|-----------|----|----|----|----|----|----|
| June 2016 |    |    |    |    |    |    |
| Su        | Mo | Tu | We | Th | Fr | Sa |
|           |    |    |    |    | 1  | 2  |
|           |    |    |    |    | 3  | 4  |
|           |    | 5  | 6  | 7  | 8  | 9  |
|           |    | 10 | 11 | 12 | 13 | 14 |
|           |    | 15 | 16 | 17 | 18 | 19 |
|           |    | 20 | 21 | 22 | 23 | 24 |
|           |    | 25 | 26 | 27 | 28 | 29 |
|           |    | 30 |    |    |    |    |

To find out the weekday of the first day of the month, call

```
int weekday = java.time.LocalDate.of(year, month, 1).getDayOfWeek().getValue();
// 1 = Monday, 2 = Tuesday, ..., 7 = Sunday
```

Make a helper function to print the header and a helper function to print each row.

- P5.15** Write a program that reads two fractions, adds them, and prints the result so that the numerator and denominator have no common factor. For example, when adding  $\frac{3}{4}$  and  $\frac{5}{6}$ , the result is  $\frac{19}{12}$ . Use helper functions for obtaining the numerator and denominator of strings such as " $3/4$ ", and for computing the greatest common divisor of two integers.

**■■ P5.16** Write a program that prints the decimal expansion of a fraction, marking the repeated part with an R. For example, 5/2 is 2.5R0, 1/12 is 0.08R3, and 1/7 is 0.R142857.

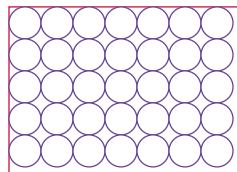
**■■ P5.17** Write a program that reads a decimal expansion with a repeated part, as in Exercise P5.16, and displays the fraction that it represents.

**■■ Graphics P5.18** Write two methods

```
public static void drawHouse(Graphics g, int x, int y)
public static void drawCar(Graphics g, int x, int y)
```

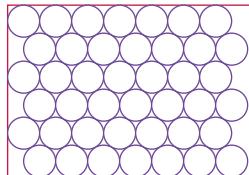
that draw a house or car at a given position. Then write a program that calls these methods to produce a suburban scene with several houses and cars.

**■■ Graphics P5.19** Write a program that reads in the width and height of a rectangle and the diameter of a circle. Then fill the rectangle with as many circles as possible, using “square circle packing”:

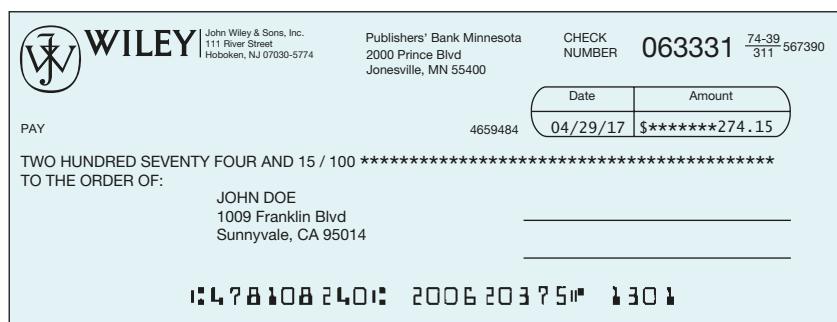


Read the inputs in the `main` method and pass them to the `draw` method. Provide a helper method to draw each row.

**■■ Graphics P5.20** Repeat Exercise P5.19 with “hexagonal circle packing”:



**■■ Business P5.21** Write a program that prints a paycheck. Ask the program user for the name of the employee, the hourly rate, and the number of hours worked. If the number of hours exceeds 40, the employee is paid “time and a half”, that is, 150 percent of the hourly rate on the hours exceeding 40. Your check should look similar to that in the figure below. Use fictitious names for the payer and the bank. Be sure to use stepwise refinement and break your solution into several methods. Use the `intName` method to print the dollar amount of the check.



**Business P5.22** *Postal bar codes.* For faster sorting of letters, the United States Postal Service encourages companies that send large volumes of mail to use a bar code denoting the zip code (see Figure 7).

The encoding scheme for a five-digit zip code is shown in Figure 8. There are full-height frame bars on each side. The five encoded digits are followed by a check digit, which is computed as follows: Add up all digits, and choose the check digit to make the sum a multiple of 10. For example, the zip code 95014 has a sum of 19, so the check digit is 1 to make the sum equal to 20.

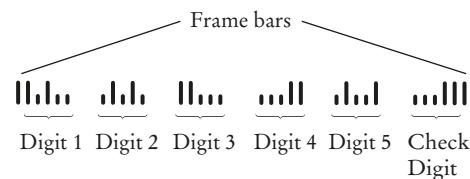
\*\*\*\*\* ECRLOT \*\* CO57

CODE C671RTS2  
JOHN DOE  
1009 FRANKLIN BLVD  
SUNNYVALE CA 95014 – 5143



**Figure 7** A Postal Bar Code

CO57



**Figure 8** Encoding for Five-Digit Bar Codes

Each digit of the zip code, and the check digit, is encoded according to the table below, where 1 denotes a full bar and 0 a half bar:

| Digit | Bar 1<br>(weight 7) | Bar 2<br>(weight 4) | Bar 3<br>(weight 2) | Bar 4<br>(weight 1) | Bar 5<br>(weight 0) |
|-------|---------------------|---------------------|---------------------|---------------------|---------------------|
| 1     | 0                   | 0                   | 0                   | 1                   | 1                   |
| 2     | 0                   | 0                   | 1                   | 0                   | 1                   |
| 3     | 0                   | 0                   | 1                   | 1                   | 0                   |
| 4     | 0                   | 1                   | 0                   | 0                   | 1                   |
| 5     | 0                   | 1                   | 0                   | 1                   | 0                   |
| 6     | 0                   | 1                   | 1                   | 0                   | 0                   |
| 7     | 1                   | 0                   | 0                   | 0                   | 1                   |
| 8     | 1                   | 0                   | 0                   | 1                   | 0                   |
| 9     | 1                   | 0                   | 1                   | 0                   | 0                   |
| 0     | 1                   | 1                   | 0                   | 0                   | 0                   |

The digit can be easily computed from the bar code using the column weights 7, 4, 2, 1, 0. For example, 01100 is  $0 \times 7 + 1 \times 4 + 1 \times 2 + 0 \times 1 + 0 \times 0 = 6$ . The only exception is 0, which would yield 11 according to the weight formula.

Write a program that asks the user for a zip code and prints the bar code. Use : for half bars, | for full bars. For example, 95014 becomes

||:|::|:|:|||:|:::|:|:|::|:||

Provide these methods:

```
public static void printDigit(int d)
public static void printBarCode(int zipCode)
```

**■■ Business P5.23** Write a program that reads in a bar code (with : denoting half bars and | denoting full bars) and prints out the zip code it represents. Print an error message if the bar code is not correct.

**■■ Business P5.24** Write a program that converts a Roman number such as MCMLXXVIII to its decimal number representation. *Hint:* First write a method that yields the numeric value of each of the letters. Then use the following algorithm:

```
total = 0
str = roman number string
While str is not empty
 If str has length 1, or value(first character of str) is at least value(second character of str)
 Add value(first character of str) to total.
 Remove first character from str.
 Else
 difference = value(second character of str) - value(first character of str)
 Add difference to total.
 Remove first character and second character from str.
```

**■■ Business P5.25** A non-governmental organization needs a program to calculate the amount of financial assistance for needy families. The formula is as follows:

- If the annual household income is between \$30,000 and \$40,000 and the household has at least three children, the amount is \$1,000 per child.
- If the annual household income is between \$20,000 and \$30,000 and the household has at least two children, the amount is \$1,500 per child.
- If the annual household income is less than \$20,000, the amount is \$2,000 per child.

Implement a method for this computation. Write a program that asks for the household income and number of children for each applicant, printing the amount returned by your method. Use -1 as a sentinel value for the input.

**■■ Business P5.26** In a social networking service, a user has friends, the friends have other friends, and so on. We are interested in knowing how many people can be reached from a person by following a given number of friendship relations. This number is called the “degree of separation”: one for friends, two for friends of friends, and so on. Because we do not have the data from an actual social network, we will simply use an average of the number of friends per user.

Write a recursive method

```
public static double reachablePeople(int degree, double averageFriendsPerUser)
```

Use that method in a program that prompts the user for the desired degree and average, and then prints the number of reachable people. This number should include the original user.



© Michael Hay/Stockphoto

**Business P5.27** Having a secure password is a very important practice, when much of our information is stored online. Write a program that validates a new password, following these rules:

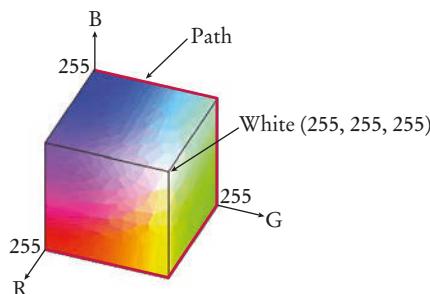
- The password must be at least 8 characters long.
- The password must have at least one uppercase and one lowercase letter
- The password must have at least one digit.

Write a program that asks for a password, then asks again to confirm it. If the passwords don't match or the rules are not fulfilled, prompt again. Your program should include a method that checks whether a password is valid.

**Science P5.28** You are designing an element for a control panel that displays a temperature value between 0 and 100. The element's color should vary continuously from blue (when the temperature is 0) to red (when the temperature is 100). Write a method `public static int colorForValue(double temperature)` that returns a color value for the given temperature. Colors are encoded as red/green/blue values, each between 0 and 255. The three colors are combined into a single integer, using the formula

$$\text{color} = 65536 \times \text{red} + 256 \times \text{green} + \text{blue}$$

Each of the intermediate colors should be fully saturated; that is, it should be on the outside of the color cube, along the path that goes from blue through cyan, green, and yellow to red.



You need to know how to *interpolate* between values. In general, if an output  $y$  should vary from  $c$  to  $d$  as an input  $x$  varies from  $a$  to  $b$ , then  $y$  is computed as follows:

$$z = (x - a) / (b - a)$$

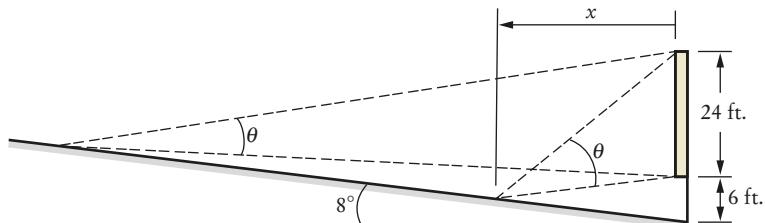
$$y = c(1 - z) + dz$$

If the temperature is between 0 and 25 degrees, interpolate between blue and cyan, whose (red, green, blue) components are  $(0, 0, 255)$  and  $(0, 255, 255)$ . For temperature values between 25 and 50, interpolate between  $(0, 255, 255)$  and  $(0, 255, 0)$ , which represents the color green. Do the same for the remaining two path segments.

You need to interpolate each color component separately and then combine the interpolated colors to a single integer.

Be sure to use appropriate helper methods to solve your task.

**Science P5.29** In a movie theater, the angle  $\theta$  at which a viewer sees the picture on the screen depends on the distance  $x$  of the viewer from the screen. For a movie theater with the dimensions shown in the picture below, write a method that computes the angle for a given distance.

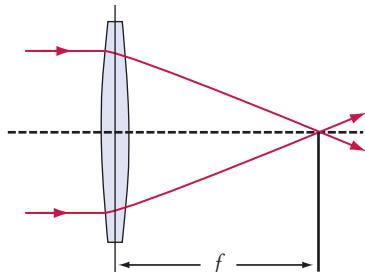


Next, provide a more general method that works for theaters with arbitrary dimensions.

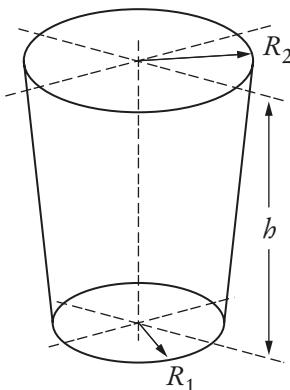
- Science P5.30** The effective focal length  $f$  of a lens of thickness  $d$  that has surfaces with radii of curvature  $R_1$  and  $R_2$  is given by

$$\frac{1}{f} = (n - 1) \left[ \frac{1}{R_1} - \frac{1}{R_2} + \frac{(n - 1)d}{nR_1R_2} \right]$$

where  $n$  is the refractive index of the lens medium. Write a method that computes  $f$  in terms of the other parameters.



- Science P5.31** A laboratory container is shaped like the frustum of a cone:



Write methods to compute the volume and surface area, using these equations:

$$V = \frac{1}{3}\pi h(R_1^2 + R_2^2 + R_1R_2)$$

$$S = \pi(R_1 + R_2)\sqrt{(R_2 - R_1)^2 + h^2} + \pi R_1^2$$

- Science P5.32** Electric wire, like that in the photo, is a cylindrical conductor covered by an insulating material. The resistance of a piece of wire is given by the formula

$$R = \frac{\rho L}{A} = \frac{4\rho L}{\pi d^2}$$

where  $\rho$  is the resistivity of the conductor, and  $L$ ,  $A$ , and  $d$  are the length, cross-sectional area, and diameter of the wire. The resistivity of copper is  $1.678 \times 10^{-8} \Omega \text{ m}$ .



The wire diameter,  $d$ , is commonly specified by the American wire gauge (AWG), which is an integer,  $n$ . The diameter of an AWG  $n$  wire is given by the formula

$$d = 0.127 \times 92^{\frac{36-n}{39}} \text{ mm}$$

Write a method

```
public static double diameter(int wireGauge)
```

that accepts the wire gauge and returns the corresponding wire diameter. Write another method

```
public static double copperWireResistance(double length, int wireGauge)
```

that accepts the length and gauge of a piece of copper wire and returns the resistance of that wire. The resistivity of aluminum is  $2.82 \times 10^{-8} \Omega \text{ m}$ . Write a third method

```
public static double aluminumWireResistance(double length, int wireGauge)
```

that accepts the length and gauge of a piece of aluminum wire and returns the resistance of that wire.

Write a program to test these methods.

**Science P5.33** The drag force on a car is given by

$$F_D = \frac{1}{2} \rho v^2 A C_D$$

where  $\rho$  is the density of air ( $1.23 \text{ kg/m}^3$ ),  $v$  is the velocity in units of  $\text{m/s}$ ,  $A$  is the projected area of the car ( $2.5 \text{ m}^2$ ), and  $C_D$  is the drag coefficient (0.2).

The amount of power in watts required to overcome such drag force is  $P = F_D v$ , and the equivalent horsepower required is  $\text{Hp} = P / 746$ . Write a program that accepts a car's velocity and computes the power in watts and in horsepower needed to overcome the resulting drag force. Note:  $1 \text{ mph} = 0.447 \text{ m/s}$ .

### ANSWERS TO SELF-CHECK QUESTIONS

1. The arguments are 3 and 2. The return value is 9.
2. The inner call to `Math.pow` returns  $2^2 = 4$ . Therefore, the outer call returns  $4^2 = 16$ .
3. 3.0
4. Users of the method can treat it as a *black box*.
5. 27
6.  $8 \times 8 \times 8 = 512$
7. 

```
double volume = Math.pow(sideLength, 3);
return volume;
```
8. 

```
public static double squareArea(
 double sideLength)
{
 double area = sideLength * sideLength;
 return area;
}
```
9.  $(2 + 3) / (3 - 2) = 5$
10. When the `mystery` method is called,  $x$  is set to 5,  $y$  is set to 7, and  $z$  becomes 12.0. Then  $z$  is changed to 6.0, and that value is returned and printed.
11. When the method is called,  $x$  is set to 5. Then  $y$  is set to 25, and that value is returned and printed.
12. When the method is called,  $n$  is set to 5. Then  $n$  is incremented twice, setting it to 7. That value is returned and printed.
13. It acts the same way: If `sideLength` is 0, it returns 0 directly instead of computing  $0 \times 0 \times 0$ .
14. It returns `true` if  $n$  is even; `false` if  $n$  is odd.

- 15.** public static boolean mystery(int n)  
 {  
     return n % 2 == 0;  
 }
- 16.** boxString("Hello");  
 boxString("World");
- 17.** The `boxString` method does not return a value. Therefore, you cannot use it in a call to the `print` method.
- 18.** public static void shout(String message)  
 {  
     System.out.println(message + "!!!!");  
 }
- 19.** public static void boxString(String contents)  
 {  
     int n = contents.length();  
     for (int i = 0; i < n + 4; i++)  
     {  
         System.out.print("-");  
     }  
     System.out.println();  
     System.out.println("!" + contents + "!");  
     for (int i = 0; i < n + 4; i++)  
     {  
         System.out.print("-");  
     }  
     System.out.println()  
 }
- 20.** public static void printLine(int count)  
 {  
     for (int i = 0; i < count; i++)  
     {  
         System.out.print("-");  
     }  
     System.out.println();  
 }  
 public static void boxString(String contents)  
 {  
     int n = contents.length();  
     printLine(n + 2);  
     System.out.println("!" + contents + "!");  
     printLine(n + 2);  
 }
- 21.** int totalPennies = getPennies(total);  
 int taxPennies = getPennies(total \* taxRate);  
 where the method is defined as  
 /\*\*  
 \* Gets a given amount in pennies.  
 \* @param amount an amount in dollars and cents  
 \* @return the number of pennies in the amount  
 \*/  
 public static int getPennies(double amount)  
 {  
     return (int) Math.round(100 \* amount) % 100;  
 }

- 22.** if (isEven(page)) . . .  
 where the method is defined as follows:  
 public static boolean isEven(int n)  
 {  
     return n % 2 == 0;  
 }
- 23.** Add parameter variables so you can pass the initial balance and interest rate to the method:  
 public static double balance(  
     double initialBalance, double rate,  
     int years)  
 {  
     return initialBalance \* pow(  
         1 + rate / 100, years);  
 }
- 24.** int spaces = countSpaces(input);  
 where the method is defined as follows:  
 /\*\*  
 \* Gets the number of spaces in a string.  
 \* @param str any string  
 \* @return the number of spaces in str  
 \*/  
 public static int countSpaces(String str)  
 {  
     int count = 0;  
     for (int i = 0; i < str.length(); i++)  
     {  
         if (str.charAt(i) == ' ')  
         {  
             count++;  
         }  
     }  
     return count;  
 }
- 25.** It is very easy to replace the space with any character.  
 /\*\*  
 \* Counts the instances of a given character  
 \* in a string.  
 \* @param str any string  
 \* @param ch a character whose occurrences  
 \* should be counted  
 \* @return the number of times that ch occurs  
 \* in str  
 \*/  
 public static int count(String str, char ch)  
 {  
     int count = 0;  
     for (int i = 0; i < str.length(); i++)  
     {  
         if (str.charAt(i) == ch) { count++; }  
     }  
     return count;  
 }

This is useful if you want to count other characters. For example, `count(input, ",")` counts the commas in the input.

- 26.** Change line 28 to

```
name = name + digitName(part / 100)
 + " hundred";
```

In line 25, add the statement

```
if (part >= 1000)
{
 name = digitName(part / 1000) + "thousand ";
 part = part % 1000;
}
```

In line 18, change 1,000 to 10,000 in the comment.

- 27.** In the case of “teens”, we already have the last digit as part of the name.

- 28.** Nothing is printed. One way of dealing with this case is to add the following statement before line 23.

```
if (number == 0) { return "zero"; }
```

- 29.** Here is the approximate trace:

| intName(number = 72) |               |
|----------------------|---------------|
| part                 | name          |
| 72                   | "seventy"     |
| 2                    | "seventy two" |

Note that the string starts with a blank space. Exercise P5.5 asks you to eliminate it.

- 30.** Here is one possible solution. Break up the task `print table` into `print header` and `print body`. The `print header` task calls `print separator`, prints the header cells, and calls `print separator` again. The `print body` task repeatedly calls `print row` and then calls `print separator`.

- 31.** Lines 14–17.

- 32.** Lines 11–19.

- 33.** The variables `x` defined in lines 5 and 15.

- 34.** Rename the local variable `x` that is declared in line 15, or rename the parameter variable `x` that is declared in line 10.

- 35.** The `main` method accesses the local variable `s` of the `mystery` method. Assuming that the `main` method intended to print the last value of `s` before the method returned, it should simply print the return value that is stored in its local variable `x`.

- 36.** `[] [] [] []`  
`[] [] []`  
`[] []`  
`[]`

- 37.**  $4 + 3 + 2 + 1 + 0 = 10$

- 38.**  $\text{mystery}(10) + 1 = \text{mystery}(5) + 2 = \text{mystery}(2) + 3 = \text{mystery}(1) + 4 = \text{mystery}(0) + 5 = 5$

- 39.** The idea is to print one `[]`, then print  $n - 1$  of them.

```
public static void printBoxes(int n)
{
 if (n == 0) { return; }
 System.out.print("[]");
 printBoxes(n - 1);
}
```

- 40.** Simply add the following to the beginning of the method:

```
if (part >= 1000)
{
 return intName(part / 1000) + " thousand "
 + intName(part % 1000);
}
```

## WORKED EXAMPLE 5.1

## Generating Random Passwords



**Problem Statement** Many web sites and software packages require you to create passwords that contain at least one digit and one special character. Your task is to write a program that generates such a password of a given length. The characters should be chosen randomly.

**Change Password**

To protect the security of your account, please change your password frequently.

[Learn more about Security Features and Protecting Your Account.](#)

**Choosing a Password**

When selecting your password, please keep the following in mind:

- **Length.** Use at least eight (8) characters without spaces.
- **Characters.** Use at least one letter, one number, and one special character, excluding < \ >.
- **Content.** Avoid numbers, names, or dates that are significant to you. For example, your phone number, first name, or date of birth. Try to base your password on a memory aid.

Enter your current password:

Enter your new password:

Retype your new password:

**Submit** **Cancel**

**Step 1** Describe what the method should do.

The problem description asks you to write a program, not a method. We will write a password-generating method and call it from the program's main method.

Let us be more precise about the method. It will generate a password with a given number of characters. We could include multiple digits and special characters, but for simplicity, we decide to include just one of each. We need to decide which special characters are valid. For our solution, we will use the following set:

+ - \* / ? ! @ # \$ % &

The remaining characters of the password are letters. For simplicity, we will use only lowercase letters in the English alphabet.

**Step 2** Determine the method's "inputs".

There is just one parameter: the length of the password.

At this point, we have enough information to document the method:

```
/**
 * Generates a random password.
 * @param length the length of the password
 * @return a password of the given length with one digit and one
 * special symbol
 */
```

**Step 3** Determine the types of the parameter variables and the return value.

The parameter is an integer, and the method returns the password, that is, a String. The method will be declared as

```
public static String makePassword(int length)
```

**Step 4** Write pseudocode for obtaining the desired result.

Here is one approach for making a password:

**Make an empty string called password.**  
**Randomly generate length - 2 letters and append them to password.**

## WE2 Chapter 5 Methods

**Randomly generate a digit and insert it at a random location in password.**  
**Randomly generate a symbol and insert it at a random location in password.**

How do we generate a random letter, digit, or symbol? How do we insert a digit or symbol in a random location? In the spirit of stepwise refinement, we will delegate those tasks to helper methods. Each of those methods starts a new sequence of steps, which, for greater clarity, we will place after the steps for this method.

### Step 5 Implement the method body.

We need to know the “black box” descriptions of the two helper methods described in Step 4 (which we will complete after this method). Here they are:

```
/**
 * Returns a string containing one character randomly chosen from a given string.
 * @param characters the string from which to randomly choose a character
 * @return a substring of length 1, taken at a random index
 */
public static String randomCharacter(String characters)

/**
 * Inserts one string into another at a random position.
 * @param str the string into which another string is inserted
 * @param toInsert the string to be inserted
 * @return the result of inserting toInsert into str
 */
public static String insertAtRandom(String str, String toInsert)
```

Now we can translate the pseudocode of Step 4 into Java:

```
public static String makePassword(int length)
{
 String password = "";
 for (int i = 0; i < length - 2; i++)
 {
 password = password + randomCharacter("abcdefghijklmnopqrstuvwxyz");
 }
 String randomDigit = randomCharacter("0123456789");
 password = insertAtRandom(password, randomDigit);
 String randomSymbol = randomCharacter("+-*/?!@#$%&");
 password = insertAtRandom(password, randomSymbol);
 return password;
}
```

### Step 6 Test your method.

Because our method depends on several helper methods, we must implement the helper methods first, as described in the following sections. (If you are impatient, you can use the technique of stubs that is described in Programming Tip 5.5.)

Here is a simple `main` method that calls the `makePassword` method:

```
public static void main(String[] args)
{
 String result = makePassword(8);
 System.out.println(result);
}
```

Place all methods into a class `PasswordGenerator`. Run the program a few times. Typical outputs are

```
u@taqr8f
i?fs1dgh
ot$3rvdv
```

Each output has length 8 and contains a digit and special symbol.

## Repeat for the First Helper Method

Now it is time to turn to the helper method for generating a random letter, digit, or special symbol.

**Step 1** Describe what the method should do.

How do we deal with the choice between letter, digit, or special symbol? Of course, we could write three separate methods, but it is better if we can solve all three tasks with a single method. We could require a parameter, such as 1 for letter, 2 for digit, and 3 for special symbol. But stepping back a bit, we can supply a more general method that simply selects a random character from *any* set. Passing the string "abcdefghijklmnopqrstuvwxyz" generates a random lowercase letter. To get a random digit, pass the string "0123456789" instead.

Now we know what our method should do. Given any string, it should return a random character in it.

**Step 2** Determine the method's "inputs".

The input is any string.

**Step 3** Determine the types of the parameter variables and the return value.

The input type is clearly `String`. We have a choice for the result. We can return a `char` value or a `String` of length 1.

We want to make a string from the random characters. It is easy to concatenate strings, so we choose to return strings of length 1.

The method will be declared as

```
public static String randomCharacter(String characters)
```

**Step 4** Write pseudocode for obtaining the desired result.

```
randomCharacter(characters)
n = length of characters
r = a random integer between 0 and n - 1
Return the substring of characters of length 1 that starts at r.
```

**Step 5** Implement the method body.

Simply translate the pseudocode into Java:

```
public static String randomCharacter(String characters)
{
 int n = characters.length();
 int r = (int) (n * Math.random());
 return characters.substring(r, r + 1);
}
```

**Step 6** Test your method.

Supply a class for testing this method only:

```
public class RandomCharacterTester
{
 public static void main(String[] args)
 {
 for (int i = 1; i <= 10; i++)
 {
 System.out.print(randomCharacter("abcdef"));
 }
 System.out.println();
 }
}
```

## WE4 Chapter 5 Methods

```
/***
 * Returns a string containing one character randomly chosen from a given string.
 * @param characters the string from which to randomly choose a character
 * @return a substring of length 1, taken at a random index
 */
public static String randomCharacter(String characters)
{
 int n = characters.length();
 int r = (int) (n * Math.random());
 return characters.substring(r, r + 1);
}
```

When you run this program, you might get an output such as

afcdfeefac

This confirms that the method works correctly.

### Repeat for the Second Helper Method

Finally, we implement the second helper method, which inserts a string containing a single character at a random location in a string.

**Step 1** Describe what the method should do.

Suppose we have a string "arxcsw" and a string "8". Then the second string should be inserted at a random location, returning a string such as "ar8xcsw" or "arxcsw8". Actually, it doesn't matter that the second string has length 1, so we will simply specify that our method should insert an arbitrary string into a given string.

**Step 2** Determine the method's "inputs".

The first input is the string into which another string should be inserted. The second input is the string to be inserted.

**Step 3** Determine the types of the parameter variables and the return value.

The inputs are both of type `String`, and the result is also a `String`. We can now fully describe our method:

```
/***
 * Inserts one string into another at a random position.
 * @param str the string into which another string is inserted
 * @param toInsert the string to be inserted
 * @return the result of inserting toInsert into str
 */
public static String insertAtRandom(String str, String toInsert)
```

**Step 4** Write pseudocode for obtaining the desired result.

There is no predefined method for inserting a string into another. Instead, we need to find the insertion position and then "break up" the first string by taking two substrings: the characters up to the insertion position, and the characters following it.

How many choices are there for the insertion position? If `str` has length 6, there are seven choices:

1. |arxcsw
2. a|rxcsw
3. ar|xcs
4. arx|cs
5. arxc|sw
6. arxcs|w
7. arxcs|w

In general, if the string has length  $n$ , there are  $n + 1$  choices, ranging from 0 (before the start of the string) to  $n$  (after the end of the string). Here is the pseudocode:

```
insertAtRandom(str, toInsert)
n = length of str
r = a random integer between 0 and n
Return the substring of str from 0 to r - 1 + toInsert + the remainder of str.
```

### Step 5

Implement the method body.

Translate the pseudocode into Java:

```
public static String insertAtRandom(String str, String toInsert)
{
 int n = str.length();
 int r = (int) ((n + 1) * Math.random());
 return str.substring(0, r) + toInsert + str.substring(r);
```

### Step 6

Test your method.

Supply a class for testing this method only:

```
public class RandomInsertionTester
{
 public static void main(String[] args)
 {
 for (int i = 1; i <= 10; i++)
 {
 System.out.println(insertAtRandom("arxcsw", "8"));
 }
 }

 /**
 * Inserts one string into another at a random position.
 * @param str the string into which another string is inserted
 * @param toInsert the string to be inserted
 * @return the result of inserting toInsert into str
 */
 public static String insertAtRandom(String str, String toInsert)
 {
 int n = str.length();
 int r = (int) ((n + 1) * Math.random());
 return str.substring(0, r) + toInsert + str.substring(r);
 }
}
```

When you run this program, you might get an output such as

```
arxcsw8
ar8xcsw
arxc8sw
a8rxcs
arxcs
ar8xcs
arxcs
ar8xcs
a8rxcs
8arxcs
8arxcs
```

The output shows that the second string is being inserted at an arbitrary position, including the beginning and end of the first string.

See `worked_example_1/PasswordGenerator.java` in your source code for the complete program.



## WORKED EXAMPLE 5.2

## Calculating a Course Grade



**Problem Statement** Students in this course take four exams and earn a letter grade (A+, A, A-, B+, B, B-, C+, C, C-, D+, D, D-, or F) for each of them. The course grade is determined by dropping the lowest grade and averaging the three remaining grades.

To average grades, first convert them to number grades, using the usual scheme A+ = 4.3, A = 4.0, A- = 3.7, B+ = 3.3, . . . , D- = 0.7, F = 0. Then compute their average and convert it back to the closest letter grade. For example, an average of 3.51 would be an A-.

Your task is to read inputs of the form:

*letterGrade1 letterGrade2 letterGrade3 letterGrade4*

For example,

A- B+ C A

For each input line, your output should be

*letterGrade*

where the letter grade is the grade earned in the course, as just described. For example,

A-

The end of inputs will be indicated by a *letterGrade1* input of Q.

### Step 1

Carry out stepwise refinement.

We will use the process of stepwise refinement. To process the inputs, we can process each line individually. Therefore, we declare a task **Process line**.

To process a line, we read the first grade and bail out if it is a Q. Otherwise, we read the four grades. Because we need them in their numeric form, we identify a task **Convert letter grade to number**.

We then have four numbers and need to find the smallest one. That is another task, **Find smallest of four numbers**. To average the remaining ones, we compute the sum of all values, subtract the smallest, and divide by three. Let's say that is not worth making into a subtask.

Next, we need to convert the result back into a letter grade. That is yet another subtask **Convert number grade to letter**. Finally, we print the letter grade. That is again so simple that it requires no subtask.

### Step 2

Convert letter grade to number.

How do we convert a letter grade to a number? Follow this algorithm:

Grade to number(grade)

ch = first character of grade

If ch is A / B / C / D / F

    Set result to 4 / 3 / 2 / 1 / 0.

    If the second character of grade is +

        Add 0.3 to result.

    If the second character of grade is -

        Subtract 0.3 from result.

    Return result.

Here is a method for that task:

```
/**
 * Converts a letter grade to a number.
 * @param grade a letter grade (A+, A, A-, . . . , D-, F)
 * @return the equivalent number grade
 */
```

## WE8 Chapter 5 Methods

```
public static double gradeToNumber(String grade)
{
 double result = 0;
 String first = grade.substring(0, 1);
 if (first.equals("A")) { result = 4; }
 else if (first.equals("B")) { result = 3; }
 else if (first.equals("C")) { result = 2; }
 else if (first.equals("D")) { result = 1; }
 if (grade.length() > 1)
 {
 String second = grade.substring(1, 2);
 if (second.equals("+"))
 {
 result = result + 0.3;
 }
 else if (second.equals("-"))
 {
 result = result - 0.3;
 }
 }
 return result;
}
```

### Step 3 Convert number grade to letter.

How do we do the opposite conversion? Here, the challenge is that we need to convert to the *nearest* letter grade. For example, if  $x$  is the number grade, then we have:

$2.5 \leq x < 2.85$ : B-

$2.85 \leq x < 3.15$ : B

$3.15 \leq x < 3.5$ : B+

We can make a method with 13 branches, one for each valid letter grade.

```
/**
 * Converts a number to the nearest letter grade.
 * @param x a number between 0 and 4.3
 * @return the nearest letter grade
 */
public static String numberToGrade(double x)
{
 if (x >= 4.15) { return "A+"; }
 if (x >= 3.85) { return "A"; }
 if (x >= 3.5) { return "A-"; }
 if (x >= 3.15) { return "B+"; }
 if (x >= 2.85) { return "B"; }
 if (x >= 2.5) { return "B-"; }
 if (x >= 2.15) { return "C+"; }
 if (x >= 1.85) { return "C"; }
 if (x >= 1.5) { return "C-"; }
 if (x >= 1.15) { return "D+"; }
 if (x >= 0.85) { return "D"; }
 if (x >= 0.5) { return "D-"; }
 return "F";
}
```

Exercise P5.10 suggests an alternate approach.

### Step 4 Find the minimum of four numbers.

Finally, how do we find the smallest of four numbers? The `Math` class has a method `min` that computes the minimum of two values, but there is no standard method for computing the

minimum of four values. We will write such a method, using the mathematical fact that  $\min(a, b, c, d) = \min(\min(a, b), \min(c, d))$ .

```
/*
 Returns the smallest of four numbers.
 @param a a number
 @param b a number
 @param c a number
 @param d a number
 @return the smallest of a, b, c, and d
*/
public static double min4(double a, double b, double c, double d)
{
 return Math.min(Math.min(a, b), Math.min(c, d));
}
```

### Step 5 Process a line.

As previously described, to process a line, we follow these steps:

- Read in the four input strings.**
- Convert grades to numbers.**
- Compute the average after dropping the lowest grade.**
- Print the grade corresponding to that average.**

However, if we read the first input string and find a Q, we need to signal to the caller that we have reached the end of the input set and that no further calls should be made.

Our method will return a boolean value, true if it was successful, false if it encountered the sentinel.

```
/*
 Processes one line of input.
 @return true if the sentinel was not encountered
*/
public static boolean processLine()
{
 System.out.print("Enter four grades or Q to quit: ");
 Scanner in = new Scanner(System.in);

 // Read four grades

 String g1 = in.next();
 if (g1.equals("Q")) { return false; }
 String g2 = in.next();
 String g3 = in.next();
 String g4 = in.next();

 // Compute and print their average

 double x1 = gradeToNumber(g1);
 double x2 = gradeToNumber(g2);
 double x3 = gradeToNumber(g3);
 double x4 = gradeToNumber(g4);
 double xlow = min4(x1, x2, x3, x4);
 double avg = (x1 + x2 + x3 + x4 - xlow) / 3;
 System.out.println(numberToGrade(avg));
 return true;
}
```

**Step 6** Write a class and the `main` method.

The `main` method is now utterly trivial. We keep calling `processLine` while it returns true.

```
public static void main(String[] args)
{
 while (processLine())
 {
 }
}
```

We place all methods into a class `Grades`. See `worked_example_2/Grades.java` in your source code for the complete program.

---

# ARRAYS AND ARRAY LISTS

## CHAPTER GOALS

To collect elements using arrays and array lists

To use the enhanced for loop for traversing arrays and array lists

To learn common algorithms for processing arrays and array lists

To work with two-dimensional arrays



© traveler111/iStockphoto.

## CHAPTER CONTENTS

### 6.1 ARRAYS 262

**SYN** Arrays 263

**CE1** Bounds Errors 267

**CE2** Uninitialized Arrays 267

**PT1** Use Arrays for Sequences of Related Items 268

**C&S** Computer Viruses 268

### 6.2 THE ENHANCED FOR LOOP 269

**SYN** The Enhanced for Loop 270

### 6.3 COMMON ARRAY ALGORITHMS 270

**CE3** Underestimating the Size of a Data Set 279

**ST1** Sorting with the Java Library 279

**ST2** Binary Search 279

### 6.4 USING ARRAYS WITH METHODS 280

**ST3** Methods with a Variable Number of Parameters 284

### 6.5 PROBLEM SOLVING: ADAPTING ALGORITHMS 284

**PT2** Reading Exception Reports 286

**HT1** Working with Arrays 287

**WE1** Rolling the Dice

### 6.6 PROBLEM SOLVING: DISCOVERING ALGORITHMS BY MANIPULATING PHYSICAL OBJECTS 291

**VE1** Removing Duplicates from an Array

### 6.7 TWO-DIMENSIONAL ARRAYS 294

**SYN** Two-Dimensional Array Declaration 295

**WE2** A World Population Table

**ST4** Two-Dimensional Arrays with Variable Row Lengths 300

**ST5** Multidimensional Arrays 301

### 6.8 ARRAY LISTS 301

**SYN** Array Lists 302

**CE4** Length and Size 311

**ST6** The Diamond Syntax 311

**VE2** Game of Life



© traveler1116/iStockphoto.

In many programs, you need to collect large numbers of values. In Java, you use the array and array list constructs for this purpose. Arrays have a more concise syntax, whereas array lists can automatically grow to any desired size. In this chapter, you will learn about arrays, array lists, and common algorithms for processing them.

## 6.1 Arrays

We start this chapter by introducing the array data type. Arrays are the fundamental mechanism in Java for collecting multiple values. In the following sections, you will learn how to declare arrays and how to access array elements.

### 6.1.1 Declaring and Using Arrays

Suppose you write a program that reads a sequence of values and prints out the sequence, marking the largest value, like this:

```
32
54
67.5
29
35
80
115 <= largest value
44.5
100
65
```

You do not know which value to mark as the largest one until you have seen them all. After all, the last value might be the largest one. Therefore, the program must first store all values before it can print them.

Could you simply store each value in a separate variable? If you know that there are ten values, then you could store the values in ten variables `value1`, `value2`, `value3`, ..., `value10`. However, such a sequence of variables is not very practical to use. You would have to write quite a bit of code ten times, once for each of the variables. In Java, an **array** is a much better choice for storing a sequence of values of the same type.

Here we create an array that can hold ten values of type `double`:

```
new double[10]
```

The number of elements (here, 10) is called the *length* of the array.

The `new` operator constructs the array. You will want to store the array in a variable so that you can access it later.

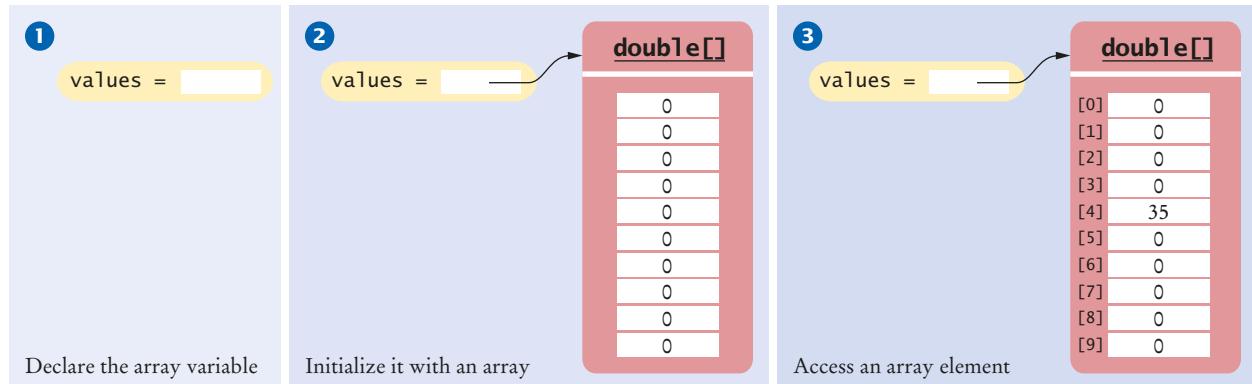
The type of an array variable is the type of the element to be stored, followed by `[]`. In this example, the type is `double[]`, because the element type is `double`.

Here is the declaration of an array variable of type `double[]` (see Figure 1):

```
double[] values; ①
```

When you declare an array variable, it is not yet initialized. You need to initialize the variable with the array:

```
double[] values = new double[10]; ②
```

**Figure 1** An Array of Size 10

Now `values` is initialized with an array of 10 numbers. By default, each number in the array is 0.

When you declare an array, you can specify the initial values. For example,

```
double[] moreValues = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```

When you supply initial values, you don't use the `new` operator. The compiler determines the length of the array by counting the initial values.

To access a value in an array, you specify which "slot" you want to use. That is done with the `[]` operator:

```
values[4] = 35; ③
```

Now the number 4 slot of `values` is filled with 35 (see Figure 1). This "slot number" is called an *index*. Each slot in an array contains an *element*.

Because `values` is an array of double values, each element `values[i]` can be used like any variable of type `double`. For example, you can display the element with index 4 with the following command:

```
System.out.println(values[4]);
```

Individual elements in an array are accessed by an integer index *i*, using the notation `array[i]`.

An array element can be used like any variable.

## Syntax 6.1

### Arrays

#### Syntax

To construct an array:

`new typeName[length]`

To access an element:

`arrayReference[index]`

Name of array variable  
Type of array variable

`double[] values = new double[10];`

Element type  
Length

`double[] moreValues = { 32, 54, 67.5, 29, 35 };`

Use brackets to access an element.

`values[i] = 0;`

List of initial values

The index must be  $\geq 0$  and  $<$  the length of the array.  
See Common Error 6.1.

Before continuing, we must take care of an important detail of Java arrays. If you look carefully at Figure 1, you will find that the *fifth* element was filled when we changed `values[4]`. In Java, the elements of arrays are numbered *starting at 0*. That is, the legal elements for the `values` array are

- `values[0]`, the first element
- `values[1]`, the second element
- `values[2]`, the third element
- `values[3]`, the fourth element
- `values[4]`, the fifth element
- ...
- `values[9]`, the tenth element

In other words, the declaration

```
double[] values = new double[10];
```

creates an array with ten elements. In this array, an index can be any integer ranging from 0 to 9.

You have to be careful that the index stays within the valid range. Trying to access an element that does not exist in the array is a serious error. For example, if `values` has ten elements, you are not allowed to access `values[20]`. Attempting to access an element whose index is not within the valid index range is called a **bounds error**. The compiler does not catch this type of error. When a bounds error occurs at run time, it causes a run-time exception.

Here is a very common bounds error:

```
double[] values = new double[10];
values[10] = value;
```

There is no `values[10]` in an array with ten elements—the index can range from 0 to 9.

To avoid bounds errors, you will want to know how many elements are in an array. The expression `values.length` yields the length of the `values` array. Note that there are no parentheses following `length`.

An array index must be at least zero and less than the size of the array.

A bounds error, which occurs if you supply an invalid array index, can cause your program to terminate.



© Luckie8/  
iStockphoto.

Like a mailbox that is identified by a box number, an array element is identified by an index.

**Table 1 Declaring Arrays**

|                                                                                                                               |                                                                                                                 |
|-------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>int[] numbers = new int[10];</code>                                                                                     | An array of ten integers. All elements are initialized with zero.                                               |
| <code>final int LENGTH = 10;</code><br><code>int[] numbers = new int[LENGTH];</code>                                          | It is a good idea to use a named constant instead of a “magic number”.                                          |
| <code>int length = in.nextInt();</code><br><code>double[] data = new double[length];</code>                                   | The length need not be a constant.                                                                              |
| <code>int[] squares = { 0, 1, 4, 9, 16 };</code>                                                                              | An array of five integers, with initial values.                                                                 |
| <code>String[] friends = { "Emily", "Bob", "Cindy" };</code>                                                                  | An array of three strings.                                                                                      |
|  <code>double[] data = new int[10];</code> | <b>Error:</b> You cannot initialize a <code>double[]</code> variable with an array of type <code>int[]</code> . |

Use the expression `array.length` to find the number of elements in an array.

The following code ensures that you only access the array when the index variable `i` is within the legal bounds:

```
if (0 <= i && i < values.length) { values[i] = value; }
```

Arrays suffer from a significant limitation: *their length is fixed*. If you start out with an array of 10 elements and later decide that you need to add additional elements, then you need to make a new array and copy all elements of the existing array into the new array. We will discuss this process in detail in Section 6.3.9.

To visit all elements of an array, use a variable for the index. Suppose `values` has ten elements and the integer variable `i` is set to 0, 1, 2, and so on, up to 9. Then the expression `values[i]` yields each element in turn. For example, this loop displays all elements in the `values` array.

```
for (int i = 0; i < 10; i++)
{
 System.out.println(values[i]);
}
```

Note that in the loop condition the index is *less than* 10 because there is no element corresponding to `values[10]`.

## 6.1.2 Array References

If you look closely at Figure 1, you will note that the variable `values` does not store any numbers. Instead, the array is stored elsewhere and the `values` variable holds a **reference** to the array. (The reference denotes the location of the array in memory.) When you access the elements in an array, you need not be concerned about the fact that Java uses array references. This only becomes important when copying array references.

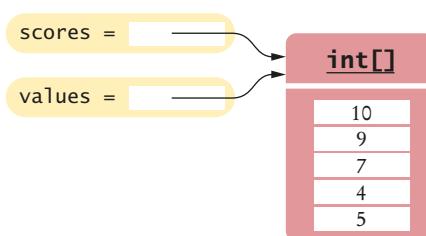
When you copy an array variable into another, both variables refer to the same array (see Figure 2).

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores; // Copying array reference
```

You can modify the array through either of the variables:

```
scores[3] = 10;
System.out.println(values[3]); // Prints 10
```

Section 6.3.9 shows how you can make a copy of the *contents* of the array.



**Figure 2**

Two Array Variables Referencing the Same Array

An array reference specifies the location of an array. Copying the reference yields a second reference to the same array.



With a partially filled array, you need to remember how many elements are filled.

With a partially filled array, keep a companion variable for the current size.

### 6.1.3 Partially Filled Arrays

An array cannot change size at run time. This is a problem when you don't know in advance how many elements you need. In that situation, you must come up with a good guess on the maximum number of elements that you need to store. For example, we may decide that we sometimes want to store more than ten elements, but never more than 100:

```
final int LENGTH = 100;
double[] values = new double[LENGTH];
```

In a typical program run, only a part of the array will be occupied by actual elements. We call such an array a **partially filled array**. You must keep a *companion variable* that counts how many elements are actually used. In Figure 3 we call the companion variable `currentSize`.

The following loop collects inputs and fills up the `values` array:

```
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
 if (currentSize < values.length)
 {
 values[currentSize] = in.nextDouble();
 currentSize++;
 }
}
```

At the end of this loop, `currentSize` contains the actual number of elements in the array. Note that you have to stop accepting inputs if the `currentSize` companion variable reaches the array length.

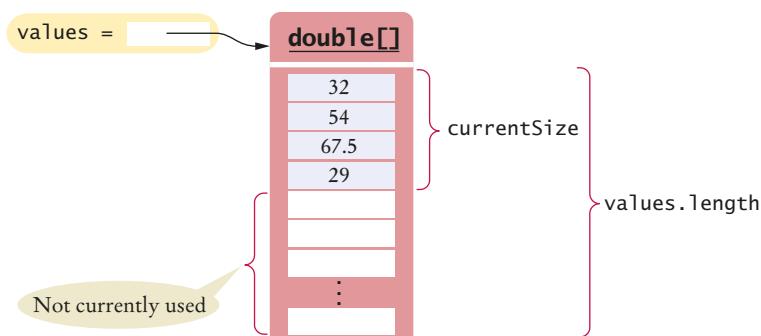
To process the gathered array elements, you again use the companion variable, not the array length. This loop prints the partially filled array:

```
for (int i = 0; i < currentSize; i++)
{
 System.out.println(values[i]);
}
```



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program demonstrating array operations.



**Figure 3** A Partially Filled Array

**SELF CHECK**

1. Declare an array of integers containing the first five prime numbers.
2. Assume the array `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?  

```
for (int i = 0; i < 2; i++)
{
 primes[4 - i] = primes[i];
}
```
3. Assume the array `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?  

```
for (int i = 0; i < 5; i++)
{
 primes[i]++;
}
```
4. Given the declaration  
`int[] values = new int[10];`  
 write statements to put the integer 10 into the elements of the array `values` with the lowest and the highest valid index.
5. Declare an array called `words` that can hold ten elements of type `String`.
6. Declare an array containing two strings, "Yes", and "No".
7. Can you produce the output on page 262 without storing the inputs in an array, by using an algorithm similar to that for finding the maximum in Section 4.7.5?

**Practice It** Now you can try these exercises at the end of the chapter: R6.2, R6.3, R6.7, E6.1.

**Common Error 6.1****Bounds Errors**

Perhaps the most common error in using arrays is accessing a nonexistent element.

```
double[] values = new double[10];
values[10] = 5.4;
// Error—values has 10 elements, and the index can range from 0 to 9
```

If your program accesses an array through an out-of-bounds index, there is no compiler error message. Instead, the program will generate an exception at run time.

**Common Error 6.2****Uninitialized Arrays**

A common error is to allocate an array variable, but not an actual array.

```
double[] values;
values[0] = 29.95; // Error—values not initialized
```

The Java compiler will catch this error. The remedy is to initialize the variable with an array:

```
double[] values = new double[10];
```

## Programming Tip 6.1



## Use Arrays for Sequences of Related Items

Arrays are intended for storing sequences of values with the same meaning. For example, an array of test scores makes perfect sense:

```
int[] scores = new int[NUMBER_OF_SCORES];
```

But an array

```
int[] personalData = new int[3];
```

that holds a person's age, bank balance, and shoe size in positions 0, 1, and 2 is bad design. It would be tedious for the programmer to remember which of these data values is stored in which array location. In this situation, it is far better to use three separate variables.



## Computing &amp; Society 6.1 Computer Viruses

In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected a significant fraction of computers connected to the Internet (which was much smaller then than it is now). Morris was sentenced to three years probation, 400 hours of community service, and a \$10,000 fine.

In order to attack a computer, a virus has to find a way to get its instructions executed. This particular program carried out a "buffer overrun" attack, providing an unexpectedly large input to a

program on another machine. That program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, that program was written in the C programming language. C, unlike Java, does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. C programmers are supposed to provide safety checks, but that had not happened in the program under attack. The virus program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes overwrote a return address, which the attacker knew was stored just after the array. When the method that read the input was finished, it didn't return to its caller but to code supplied by the virus (see the figure). The virus was thus able to execute its code on a remote machine and infect it.

In Java, as in C, programmers must be careful not to overrun array boundaries. However, in Java, this error causes a run-time exception, and it never corrupts memory outside the array. This is one of the safety features of Java.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often take permanent residence in the attacked computers. Criminal enterprises rent out the processing power of millions of hijacked computers for sending spam e-mail. Other viruses monitor every keystroke and send those that look like

credit card numbers or banking passwords to their master.

Typically, a machine gets infected because a user executes code downloaded from the Internet, clicking on an icon or link that purports to be a game or video clip. Antivirus programs check all downloaded programs against an ever-growing list of known viruses.

When you use a computer for managing your finances, you need to be aware of the risk of infection. If a virus reads your banking password and empties your account, you will have a hard time convincing your financial institution that it wasn't your act, and you will most likely lose your money. It is a good idea to use banks that require "two-factor authentication" for major transactions, such as a callback on your cell phone.

Viruses are even used for military purposes. In 2010, a virus dubbed Stuxnet spread through Microsoft Windows and infected USB sticks. The virus looked for Siemens industrial computers and reprogrammed them in subtle ways. It appears that the virus was designed to damage the centrifuges of the Iranian nuclear enrichment operation. The computers controlling the centrifuges were not connected to the Internet, but they were configured with USB sticks, some of which were infected. Security researchers believe that the virus was developed by U.S. and Israeli intelligence agencies, and that it was successful in slowing down the Iranian nuclear program. Neither country has officially acknowledged or denied their role in the attacks.

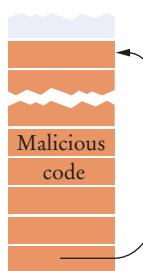
## 1 Before the attack



Buffer for input  
(512 bytes)

Return address

## 2 After the attack



Overrun buffer  
(536 bytes)

Malicious  
code

Return address

A "Buffer Overrun" Attack

## 6.2 The Enhanced for Loop

You can use the enhanced for loop to visit all elements of an array.

Often, you need to visit all elements of an array. The *enhanced for loop* makes this process particularly easy to program.

Here is how you use the enhanced for loop to total up all elements in an array named `values`:

```
double[] values = . . .;
double total = 0;
for (double element : values)
{
 total = total + element;
}
```

The loop body is executed for each element in the array `values`. At the beginning of each loop iteration, the next element is assigned to the variable `element`. Then the loop body is executed. You should read this loop as “for each element in `values`”.

This loop is equivalent to the following for loop and an explicit index variable:

```
for (int i = 0; i < values.length; i++)
{
 double element = values[i];
 total = total + element;
}
```

Note an important difference between the enhanced for loop and the basic for loop. In the enhanced for loop, the *element variable* is assigned `values[0]`, `values[1]`, and so on. In the basic for loop, the *index variable* `i` is assigned 0, 1, and so on.

Keep in mind that the enhanced for loop has a very specific purpose: getting the elements of a collection, from the beginning to the end. It is not suitable for all array algorithms. In particular, the enhanced for loop does not allow you to modify the contents of an array. The following loop does not fill an array with zeroes:

```
for (double element : values)
{
 element = 0; // ERROR: this assignment does not modify array elements
}
```

When the loop is executed, the variable `element` is set to `values[0]`. Then `element` is set to 0, then to `values[1]`, then to 0, and so on. The `values` array is not modified. The remedy is simple: Use a basic for loop:

```
for (int i = 0; i < values.length; i++)
{
 values[i] = 0; // OK
}
```



© Steve Cole/Stockphoto.

*The enhanced for loop is a convenient mechanism for traversing all elements in a collection.*



### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program that demonstrates the enhanced for loop.

## Syntax 6.2 The Enhanced for Loop

*Syntax*

```
for (typeName variable : collection)
{
 statements
}
```

This variable is set in each loop iteration.  
It is only defined inside the loop.

These statements  
are executed for each  
element.

```
for (double element : values)
{
 sum = sum + element;
}
```

An array

The variable  
contains an element,  
not an index.

### SELF CHECK



8. What does this enhanced for loop do?

```
int counter = 0;
for (double element : values)
{
 if (element == 0) { counter++; }
}
```

9. Write an enhanced for loop that prints all elements in the array `values`.  
 10. Write an enhanced for loop that multiplies all elements in a `double[]` array named `factors`, accumulating the result in a variable named `product`.  
 11. Why is the enhanced for loop not an appropriate shortcut for the following basic for loop?

```
for (int i = 0; i < values.length; i++) { values[i] = i * i; }
```

**Practice It** Now you can try these exercises at the end of the chapter: R6.8, R6.9, R6.10.

## 6.3 Common Array Algorithms

In the following sections, we discuss some of the most common algorithms for working with arrays. If you use a partially filled array, remember to replace `values.length` with the companion variable that represents the current size of the array.

### 6.3.1 Filling

This loop fills an array with squares (0, 1, 4, 9, 16, ...). Note that the element with index 0 contains  $0^2$ , the element with index 1 contains  $1^2$ , and so on.

```
for (int i = 0; i < values.length; i++)
{
 values[i] = i * i;
}
```

### 6.3.2 Sum and Average Value

You have already encountered this algorithm in Section 4.7.1. When the values are located in an array, the code looks much simpler:

```
double total = 0;
for (double element : values)
{
 total = total + element;
}
double average = 0;
if (values.length > 0) { average = total / values.length; }
```

### 6.3.3 Maximum and Minimum

Use the algorithm from Section 4.7.5 that keeps a variable for the largest element already encountered. Here is the implementation of that algorithm for an array:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
 if (values[i] > largest)
 {
 largest = values[i];
 }
}
```

Note that the loop starts at 1 because we initialize `largest` with `values[0]`. To compute the smallest element, reverse the comparison.  
These algorithms require that the array contain at least one element.

### 6.3.4 Element Separators

When separating elements, don't place a separator before the first element.

When you display the elements of an array, you usually want to separate them, often with commas or vertical lines, like this:

32 | 54 | 67.5 | 29 | 35

Note that there is one fewer separator than there are numbers. Print the separator before each element in the sequence *except the initial one* (with index 0) like this:

```
for (int i = 0; i < values.length; i++)
{
 if (i > 0)
 {
 System.out.print(" | ");
 }
 System.out.print(values[i]);
}
```

If you want comma separators, you can use the `Arrays.toString` method. The expression

`Arrays.toString(values)`

returns a string describing the contents of the array `values` in the form

[32, 54, 67.5, 29, 35]



© CEFutcher/iStockphoto.



To print five elements, you need four separators.

The elements are surrounded by a pair of brackets and separated by commas. This method can be convenient for debugging:

```
System.out.println("values=" + Arrays.toString(values));
```

### 6.3.5 Linear Search

© yekorzh/Stockphoto.



To search for a specific element, visit the elements and stop when you encounter the match.

A linear search inspects elements in sequence until a match is found.

You often need to search for the position of a specific element in an array so that you can replace or remove it. Visit all elements until you have found a match or you have come to the end of the array. Here we search for the position of the first element in an array that is equal to 100:

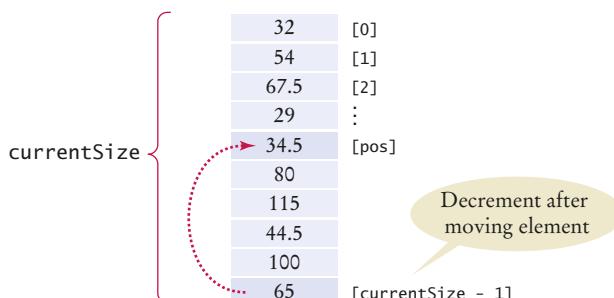
```
int searchedValue = 100;
int pos = 0;
boolean found = false;
while (pos < values.length && !found)
{
 if (values[pos] == searchedValue)
 {
 found = true;
 }
 else
 {
 pos++;
 }
}
if (found) { System.out.println("Found at position: " + pos); }
else { System.out.println("Not found"); }
```

This algorithm is called **linear search** or *sequential search* because you inspect the elements in sequence. If the array is sorted, you can use the more efficient **binary search** algorithm—see Special Topic 6.2 on page 279.

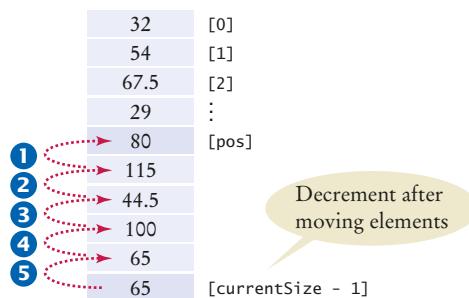
### 6.3.6 Removing an Element

Suppose you want to remove the element with index `pos` from the array `values`. As explained in Section 6.1.3, you need a companion variable for tracking the number of elements in the array. In this example, we use a companion variable called `currentSize`.

If the elements in the array are not in any particular order, simply overwrite the element to be removed with the *last* element of the array, then decrement the `currentSize` variable. (See Figure 4.)



**Figure 4**  
Removing an Element in an Unordered Array



**Figure 5**  
Removing an Element in an Ordered Array

```
values[pos] = values[currentSize - 1];
currentSize--;
```

The situation is more complex if the order of the elements matters. Then you must move all elements following the element to be removed to a lower index, and then decrement the variable holding the size of the array. (See Figure 5.)

```
for (int i = pos + 1; i < currentSize; i++)
{
 values[i - 1] = values[i];
}
currentSize--;
```

### 6.3.7 Inserting an Element

In this section, you will see how to insert an element into an array. Note that you need a companion variable for tracking the array size, as explained in Section 6.1.3.

If the order of the elements does not matter, you can simply insert new elements at the end, incrementing the variable tracking the size.

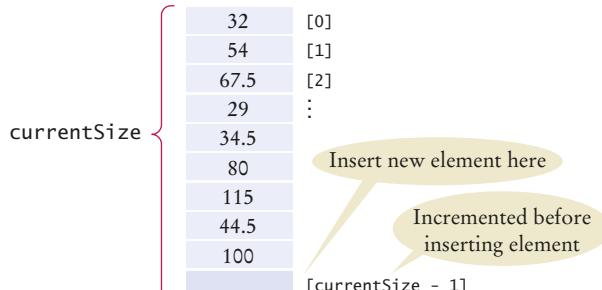
```
if (currentSize < values.length)
{
 currentSize++;
 values[currentSize - 1] = newElement;
}
```

It is more work to insert an element at a particular position in the middle of an array. First, move all elements after the insertion location to a higher index. Then insert the new element (see Figure 7).

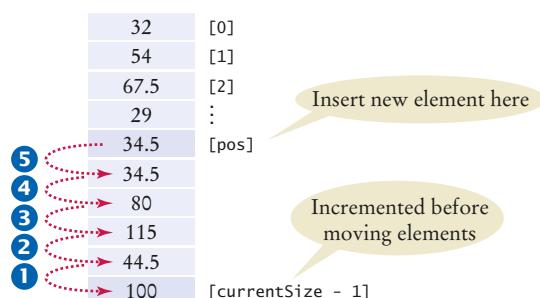
Note the order of the movement: When you remove an element, you first move the next element to a lower index, then the one after that, until you finally get to the end of the array. When you insert an element, you start at the end of the array, move that element to a higher index, then move the one before that, and so on until you finally get to the insertion location.

```
if (currentSize < values.length)
{
 currentSize++;
 for (int i = currentSize - 1; i > pos; i--)
 {
 values[i] = values[i - 1];
 }
 values[pos] = newElement;
}
```

Before inserting  
an element, move  
elements to the end  
of the array *starting*  
*with the last one*.



**Figure 6**  
Inserting an Element in an Unordered Array



**Figure 7**  
Inserting an Element in an Ordered Array

### 6.3.8 Swapping Elements

You often need to swap elements of an array. For example, you can sort an array by repeatedly swapping elements that are not in order.

Consider the task of swapping the elements at positions  $i$  and  $j$  of an array `values`. We'd like to set `values[i]` to `values[j]`. But that overwrites the value that is currently stored in `values[i]`, so we want to save that first:

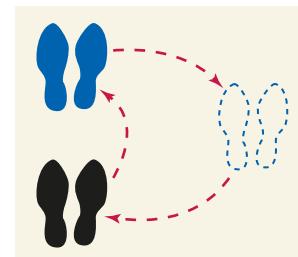
Use a temporary variable when swapping two elements.

```
double temp = values[i];
values[i] = values[j];
```

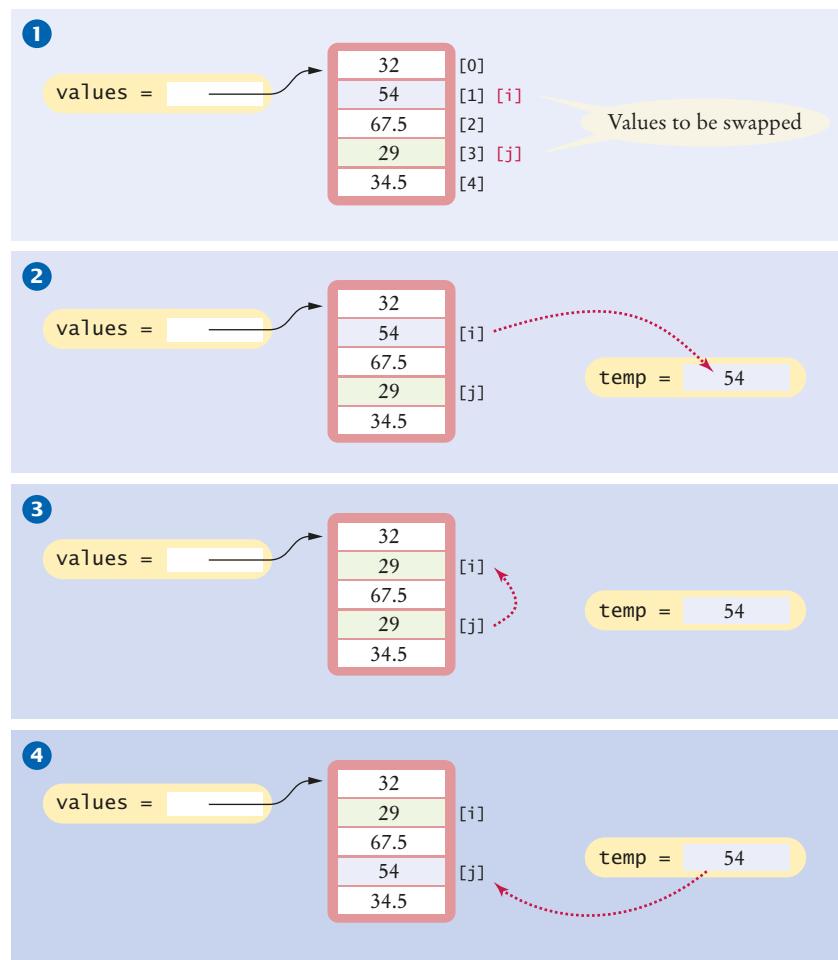
Now we can set `values[j]` to the saved value.

```
values[j] = temp;
```

Figure 8 shows the process.



*To swap two elements, you need a temporary variable.*



**Figure 8** Swapping Array Elements

### 6.3.9 Copying Arrays

Array variables do not themselves hold array elements. They hold a reference to the actual array. If you copy the reference, you get another reference to the same array (see Figure 9):

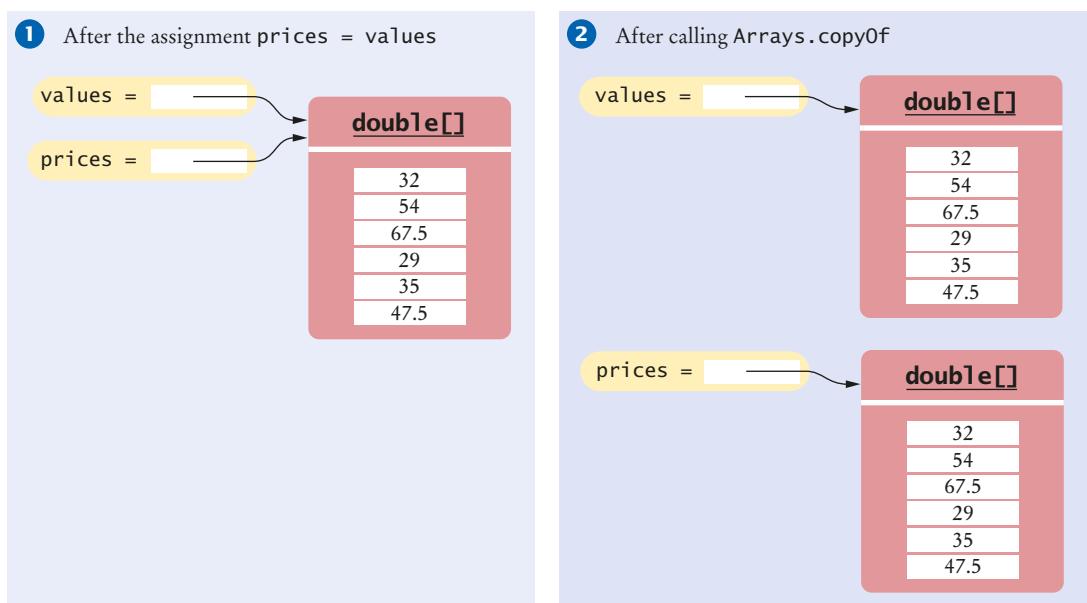
```
double[] values = new double[6];
. . . // Fill array
double[] prices = values; ①
```

Use the `Arrays.copyOf` method to copy the elements of an array into a new array.

If you want to make a true copy of an array, call the `Arrays.copyOf` method (as shown in Figure 9).

```
double[] prices = Arrays.copyOf(values, values.length); ②
```

The call `Arrays.copyOf(values, n)` allocates an array of length  $n$ , copies the first  $n$  elements of `values` (or the entire `values` array if  $n > \text{values.length}$ ) into it, and returns the new array.



**Figure 9** Copying an Array Reference versus Copying an Array

In order to use the `Arrays` class, you need to add the following statement to the top of your program:

```
import java.util.Arrays;
```

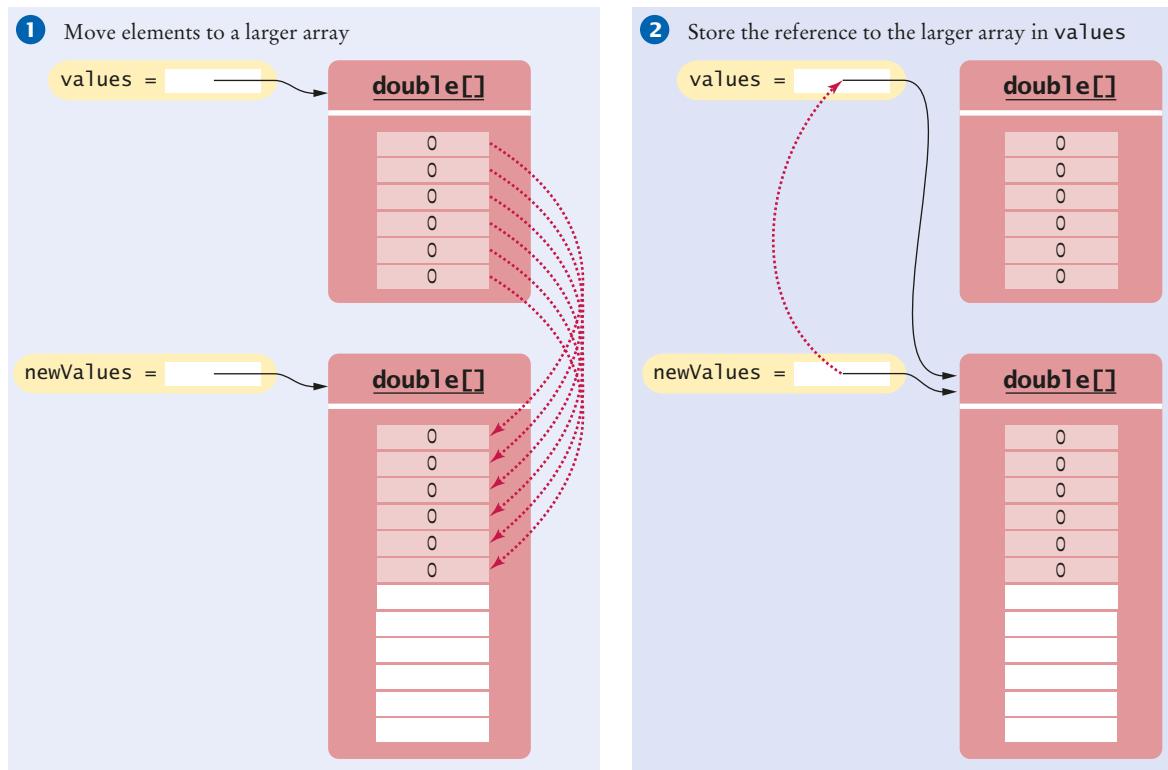
Another use for `Arrays.copyOf` is to grow an array that has run out of space. The following statements have the effect of doubling the length of an array (see Figure 10):

```
double[] newValues = Arrays.copyOf(values, 2 * values.length); ①
values = newValues; ②
```

The `copyOf` method was added in Java 6. If you use Java 5, replace

```
double[] newValues = Arrays.copyOf(values, n)
```

with

**Figure 10** Growing an Array

```
double[] newValues = new double[n];
for (int i = 0; i < n && i < values.length; i++)
{
 newValues[i] = values[i];
```

### 6.3.10 Reading Input

If you know how many inputs the user will supply, it is simple to place them into an array:

```
double[] inputs = new double[NUMBER_OF_INPUTS];
for (i = 0; i < inputs.length; i++)
{
 inputs[i] = in.nextDouble();
```

However, this technique does not work if you need to read a sequence of arbitrary length. In that case, add the inputs to an array until the end of the input has been reached.

```
int currentSize = 0;
while (in.hasNextDouble() && currentSize < inputs.length)
{
 inputs[currentSize] = in.nextDouble();
 currentSize++;
}
```

Now `inputs` is a partially filled array, and the companion variable `currentSize` is set to the number of inputs.

However, this loop silently throws away inputs that don't fit into the array. A better approach is to grow the array to hold all inputs.

```
double[] inputs = new double[INITIAL_SIZE];
int currentSize = 0;
while (in.hasNextDouble())
{
 // Grow the array if it has been completely filled
 if (currentSize >= inputs.length)
 {
 inputs = Arrays.copyOf(inputs, 2 * inputs.length); // Grow the inputs array
 }

 inputs[currentSize] = in.nextDouble();
 currentSize++;
}
```

When you are done, you can discard any excess (unfilled) elements:

```
inputs = Arrays.copyOf(inputs, currentSize);
```

The following program puts these algorithms to work, solving the task that we set ourselves at the beginning of this chapter: to mark the largest value in an input sequence.

### sec03/LargestInArray.java

```
1 import java.util.Scanner;
2
3 /**
4 This program reads a sequence of values and prints them, marking the largest value.
5 */
6 public class LargestInArray
7 {
8 public static void main(String[] args)
9 {
10 final int LENGTH = 100;
11 double[] values = new double[LENGTH];
12 int currentSize = 0;
13
14 // Read inputs
15
16 System.out.println("Please enter values, Q to quit:");
17 Scanner in = new Scanner(System.in);
18 while (in.hasNextDouble() && currentSize < values.length)
19 {
20 values[currentSize] = in.nextDouble();
21 currentSize++;
22 }
23
24 // Find the largest value
25
26 double largest = values[0];
27 for (int i = 1; i < currentSize; i++)
28 {
29 if (values[i] > largest)
30 {
31 largest = values[i];
32 }
33 }
34 }
```

```

34 // Print all values, marking the largest
35
36
37 for (int i = 0; i < currentSize; i++)
38 {
39 System.out.print(values[i]);
40 if (values[i] == largest)
41 {
42 System.out.print(" <= largest value");
43 }
44 System.out.println();
45 }
46 }
47 }
```

**Program Run**

```

Please enter values, Q to quit:
34.5 80 115 44.5 Q
34.5
80
115 <= largest value
44.5
```



- 12.** Given these inputs, what is the output of the `LargestInArray` program?  
20 10 20 Q
- 13.** Write a loop that counts how many elements in an array are equal to zero.
- 14.** Consider the algorithm to find the largest element in an array. Why don't we initialize `largest` and `i` with zero, like this?

```

double largest = 0;
for (int i = 0; i < values.length; i++)
{
 if (values[i] > largest)
 {
 largest = values[i];
 }
}
```

- 15.** When printing separators, we skipped the separator before the initial element. Rewrite the loop so that the separator is printed *after* each element, except for the last element.
- 16.** What is wrong with these statements for printing an array with separators?  
`System.out.print(values[0]);`  
`for (int i = 1; i < values.length; i++)`  
`{`  
 `System.out.print(", " + values[i]);`  
`}`

- 17.** When finding the position of a match, we used a `while` loop, not a `for` loop. What is wrong with using this loop instead?  
`for (pos = 0; pos < values.length && !found; pos++)`  
`{`  
 `if (values[pos] > 100)`  
 `{`  
 `found = true;`

```

 }
}

```

- 18.** When inserting an element into an array, we moved the elements with larger index values, starting at the end of the array. Why is it wrong to start at the insertion location, like this?

```

for (int i = pos; i < currentSize - 1; i++)
{
 values[i + 1] = values[i];
}

```

**Practice It** Now you can try these exercises at the end of the chapter: R6.18, R6.21, E6.13.

### Common Error 6.3



### Underestimating the Size of a Data Set

Programmers commonly underestimate the amount of input data that a user will pour into an unsuspecting program. Suppose you write a program to search for text in a file. You store each line in a string, and keep an array of strings. How big do you make the array? Surely nobody is going to challenge your program with an input that is more than 100 lines. Really? It is very easy to feed in the entire text of *Alice in Wonderland* or *War and Peace* (which are available on the Internet). All of a sudden, your program has to deal with tens or hundreds of thousands of lines. You either need to allow for large inputs or politely reject the excess input.

### Special Topic 6.1



### Sorting with the Java Library

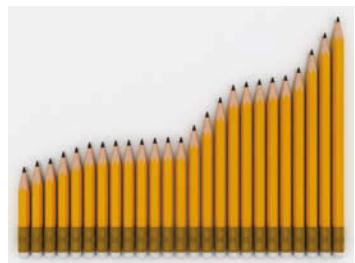
Sorting an array efficiently is not an easy task. You will learn in Chapter 14 how to implement efficient sorting algorithms. Fortunately, the Java library provides an efficient `sort` method.

To sort an array `values`, call

```
Arrays.sort(values);
```

If the array is partially filled, call

```
Arrays.sort(values, 0, currentSize);
```



© ProstoVova/iStockphoto.

### Special Topic 6.2



### Binary Search

When an array is sorted, there is a much faster search algorithm than the linear search of Section 6.3.5.

Consider the following sorted array `values`.

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

We would like to see whether the number 15 is in the array. Let's narrow our search by finding whether the number is in the first or second half of the array. The last point in the first half of the `values` array, `values[4]`, is 9, which is smaller than the number we are looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

The middle element of this sequence is 20; hence, the number must be located in the sequence:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

The last element of the first half of this very short sequence is 12, which is smaller than the number that we are searching, so we must look in the second half:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
| 1   | 4   | 5   | 8   | 9   | 12  | 17  | 20  | 24  | 32  |

We still don't have a match because  $15 \neq 17$ , and we cannot divide the subsequence further. If we wanted to insert 15 into the sequence, we would need to insert it just before values[6].

This search process is called a **binary search**, because we cut the size of the search in half in each step. That cutting in half works only because we know that the array is sorted. Here is an implementation in Java:

```
boolean found = false;
int low = 0;
int high = values.length - 1;
int pos = 0;
while (low <= high && !found)
{
 pos = (low + high) / 2; // Midpoint of the subsequence
 if (values[pos] == searchedNumber) { found = true; }
 else if (values[pos] < searchedNumber) { low = pos + 1; } // Look in second half
 else { high = pos - 1; } // Look in first half
}
if (found) { System.out.println("Found at position " + pos); }
else { System.out.println("Not found. Insert before position " + pos); }
```

## 6.4 Using Arrays with Methods

Arrays can occur as method arguments and return values.

In this section, we will explore how to write methods that process arrays.

When you define a method with an array argument, you provide a parameter variable for the array. For example, the following method computes the sum of an array of floating-point numbers:

```
public static double sum(double[] values)
{
 double total = 0;
 for (double element : values)
 {
 total = total + element;
 }
 return total;
}
```

This method visits the array elements, but it does not modify them. It is also possible to modify the elements of an array. The following method multiplies all elements of an array by a given factor:

```
public static void multiply(double[] values, double factor)
{
 for (int i = 0; i < values.length; i++)
 {
```

```

 values[i] = values[i] * factor;
 }
}

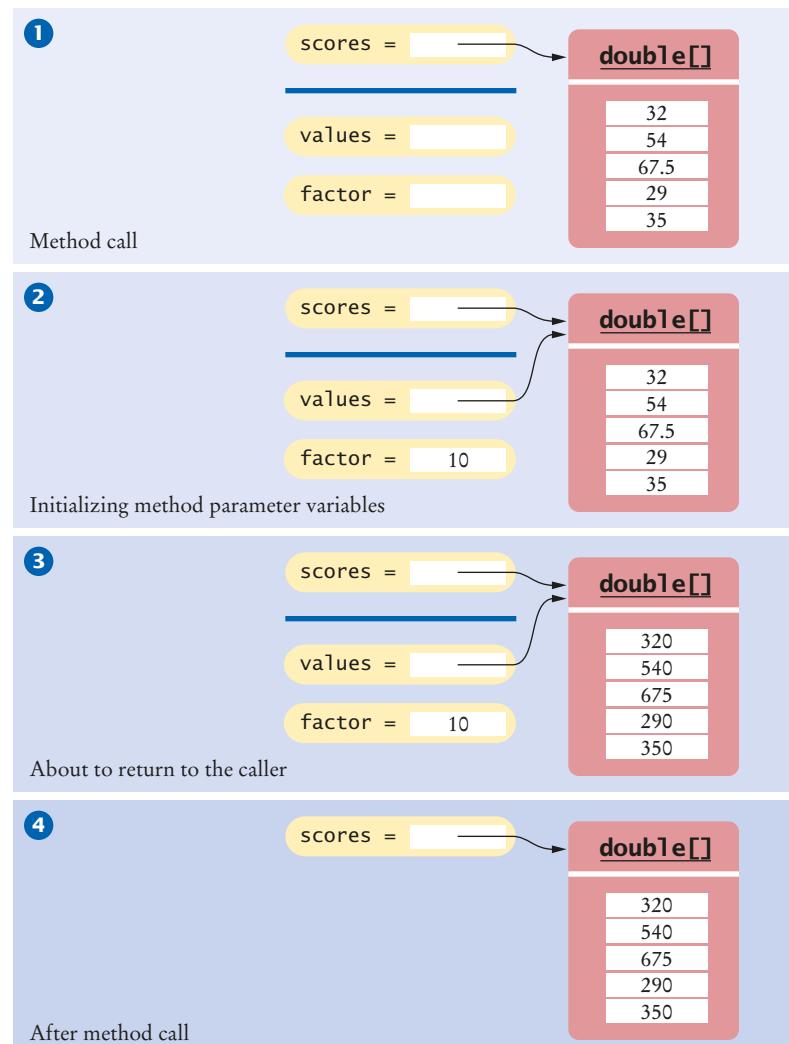
```

Figure 11 traces the method call

```
multiply(scores, 10);
```

Note these steps:

- The parameter variables `values` and `factor` are created. ①
- The parameter variables are initialized with the arguments that are passed in the call. In our case, `values` is set to `scores` and `factor` is set to 10. Note that `values` and `scores` are references to the *same* array. ②
- The method multiplies all array elements by 10. ③
- The method returns. Its parameter variables are removed. However, `scores` still refers to the array with the modified elements. ④



**Figure 11**  
Trace of Call to  
the `multiply` Method

A method can return an array. Simply build up the result in the method and return it. In this example, the squares method returns an array of squares from  $0^2$  up to  $(n - 1)^2$ :

```
public static int[] squares(int n)
{
 int[] result = new int[n];
 for (int i = 0; i < n; i++)
 {
 result[i] = i * i;
 }
 return result;
}
```

The following example program reads values from standard input, multiplies them by 10, and prints the result in reverse order. The program uses three methods:

- The `readInputs` method returns an array, using the algorithm of Section 6.3.10.
- The `multiply` method has an array argument. It modifies the array elements.
- The `printReversed` method also has an array argument, but it does not modify the array elements.

### sec04/Reverse.java

```
1 import java.util.Scanner;
2
3 /**
4 * This program reads, scales, and reverses a sequence of numbers.
5 */
6 public class Reverse
7 {
8 public static void main(String[] args)
9 {
10 double[] numbers = readInputs(5);
11 multiply(numbers, 10);
12 printReversed(numbers);
13 }
14
15 /**
16 * Reads a sequence of floating-point numbers.
17 * @param numberofInputs the number of inputs to read
18 * @return an array containing the input values
19 */
20 public static double[] readInputs(int numberofInputs)
21 {
22 System.out.println("Enter " + numberofInputs + " numbers: ");
23 Scanner in = new Scanner(System.in);
24 double[] inputs = new double[numberofInputs];
25 for (int i = 0; i < inputs.length; i++)
26 {
27 inputs[i] = in.nextDouble();
28 }
29 return inputs;
30 }
31
32 /**
33 * Multiplies all elements of an array by a factor.
34 * @param values an array
35 * @param factor the value with which element is multiplied
36 */
```

```

37 public static void multiply(double[] values, double factor)
38 {
39 for (int i = 0; i < values.length; i++)
40 {
41 values[i] = values[i] * factor;
42 }
43 }
44
45 /**
46 Prints an array in reverse order.
47 @param values an array of numbers
48 @return an array that contains the elements of values in reverse order
49 */
50 public static void printReversed(double[] values)
51 {
52 // Traverse the array in reverse order, starting with the last element
53 for (int i = values.length - 1; i >= 0; i--)
54 {
55 System.out.print(values[i] + " ");
56 }
57 System.out.println();
58 }
59 }
```

### Program Run

```

Enter 5 numbers:
12 25 20 0 10
100.0 0.0 200.0 250.0 120.0
```



#### SELF CHECK

19. How do you call the squares method to compute the first five squares and store the result in an array numbers?
20. Write a method fill that fills all elements of an array of integers with a given value. For example, the call fill(scores, 10) should fill all elements of the array scores with the value 10.
21. Describe the purpose of the following method:

```

public static int[] mystery(int length, int n)
{
 int[] result = new int[length];
 for (int i = 0; i < result.length; i++)
 {
 result[i] = (int) (n * Math.random());
 }
 return result;
}
```

22. Consider the following method that reverses an array:

```

public static int[] reverse(int[] values)
{
 int[] result = new int[values.length];
 for (int i = 0; i < values.length; i++)
 {
 result[i] = values[values.length - 1 - i];
 }
 return result;
}
```

Suppose the reverse method is called with an array scores that contains the numbers 1, 4, and 9. What is the contents of scores after the method call?

- 23.** Provide a trace diagram of the reverse method when called with an array that contains the values 1, 4, and 9.

**Practice It** Now you can try these exercises at the end of the chapter: R6.24, E6.6, E6.7.

### Special Topic 6.3



#### Methods with a Variable Number of Parameters

Starting with Java version 5.0, it is possible to declare methods that receive a variable number of parameters. For example, we can write a `sum` method that can compute the sum of any number of arguments:

```
int a = sum(1, 3); // Sets a to 4
int b = sum(1, 7, 2, 9); // Sets b to 19
```

The modified `sum` method must be declared as

```
public static void sum(int... values)
```

The `...` symbol indicates that the method can receive any number of `int` arguments. The `values` parameter variable is actually an `int[]` array that contains all arguments that were passed to the method. The method implementation traverses the `values` array and processes the elements:

```
public void sum(int... values)
{
 int total = 0;
 for (int i = 0; i < values.length; i++) // values is an int[]
 {
 total = total + values[i];
 }
 return total;
}
```

## 6.5 Problem Solving: Adapting Algorithms

By combining fundamental algorithms, you can solve complex programming tasks.

In Section 6.3, you were introduced to a number of fundamental array algorithms. These algorithms form the building blocks for many programs that process arrays. In general, it is a good problem-solving strategy to have a repertoire of fundamental algorithms that you can combine and adapt.

Consider this example problem: You are given the quiz scores of a student. You are to compute the final quiz score, which is the sum of all scores after dropping the lowest one. For example, if the scores are

8 7 8.5 9.5 7 4 10

then the final score is 50.

We do not have a ready-made algorithm for this situation. Instead, consider which algorithms may be related. These include:

- Calculating the sum (Section 6.3.2)
- Finding the minimum value (Section 6.3.3)
- Removing an element (Section 6.3.6)

We can formulate a plan of attack that combines these algorithms:

- Find the minimum.**
- Remove the minimum from the array.**
- Calculate the sum.**

Let's try it out with our example. The minimum of

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
| 8   | 7   | 8.5 | 9.5 | 7   | 4   | 10  |

is 4. How do we remove it?

Now we have a problem. The removal algorithm in Section 6.3.6 locates the element to be removed by using the *position* of the element, not the value.

But we have another algorithm for that:

- Linear search (Section 6.3.5)

We need to fix our plan of attack:

- Find the minimum value.**
- Find the position of the minimum.**
- Remove the element at the position.**
- Calculate the sum.**

Will it work? Let's continue with our example.

We found a minimum value of 4. Linear search tells us that the value 4 occurs at position 5.

|     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] |
| 8   | 7   | 8.5 | 9.5 | 7   | 4   | 10  |

We remove it:

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| [0] | [1] | [2] | [3] | [4] | [5] |
| 8   | 7   | 8.5 | 9.5 | 7   | 10  |

Finally, we compute the sum:  $8 + 7 + 8.5 + 9.5 + 7 + 10 = 50$ .

This walkthrough demonstrates that our strategy works.

Can we do better? It seems a bit inefficient to find the minimum and then make another pass through the array to obtain its position.

We can adapt the algorithm for finding the minimum to yield the position of the minimum. Here is the original algorithm:

```
double smallest = values[0];
for (int i = 1; i < values.length; i++)
{
 if (values[i] < smallest)
 {
 smallest = values[i];
 }
}
```

When we find the smallest value, we also want to update the position:

```
if (values[i] < smallest)
{
 smallest = values[i];
 smallestPosition = i;
}
```

You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

In fact, then there is no reason to keep track of the smallest value any longer. It is simply `values[smallestPosition]`. With this insight, we can adapt the algorithm as follows:

```
int smallestPosition = 0;
for (int i = 1; i < values.length; i++)
{
 if (values[i] < values[smallestPosition])
 {
 smallestPosition = i;
 }
}
```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj1o2code](http://wiley.com/go/bj1o2code) to download a program that computes the final score using the adapted algorithm for finding the minimum.

With this adaptation, our problem is solved with the following strategy:

- Find the position of the minimum.**
- Remove the element at the position.**
- Calculate the sum.**

The next section shows you a technique for discovering a new algorithm when none of the fundamental algorithms can be adapted to a task.



#### SELF CHECK

24. Section 6.3.6 has two algorithms for removing an element. Which of the two should be used to solve the task described in this section?
25. It isn't actually necessary to *remove* the minimum in order to compute the total score. Describe an alternative.
26. How can you print the number of positive and negative values in a given array, using one or more of the algorithms in Section 4.7?
27. How can you print all positive values in an array, separated by commas?
28. Consider the following algorithm for collecting all matches in an array:

```
int matchesSize = 0;
for (int i = 0; i < values.length; i++)
{
 if (values[i] fulfills the condition)
 {
 matches[matchesSize] = values[i];
 matchesSize++;
 }
}
```

How can this algorithm help you with Self Check 27?

**Practice It** Now you can try these exercises at the end of the chapter: R6.25, R6.26.

#### Programming Tip 6.2



#### Reading Exception Reports

You will sometimes have programs that terminate, reporting an “exception”, such as

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
at Homework1.processValues(Homework1.java:14)
at Homework1.main(Homework1.java:36)
```

Quite a few students give up at that point, saying “it didn’t work”, or “my program died”, without reading the error message. Admittedly, the format of the exception report is not very friendly. But, with some practice, it is easy to decipher it.

There are two pieces of useful information:

1. The name of the exception, such as `ArrayIndexOutOfBoundsException`
2. The stack trace, that is, the method calls that led to the exception, such as `Homework1.java:14` and `Homework1.java:36` in our example.

The name of the exception is always in the first line of the report, and it ends in `Exception`. If you get an `ArrayIndexOutOfBoundsException`, then there was a problem with an invalid array index. That is useful information.

To determine the line number of the offending code, look at the file names and line numbers. The first line of the stack trace is the method that actually generated the exception. The last line of the stack trace is a line in `main`. In our example, the exception was caused by line 14 of `Homework1.java`. Open up the file, go to that line, and look at it! Also look at the name of the exception. In most cases, these two pieces of information will make it completely obvious what went wrong, and you can easily fix your error.

Sometimes, the exception was thrown by a method that is in the standard library. Here is a typical example:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index
out of range: -4
at java.lang.String.substring(String.java:1444)
at Homework2.main(Homework2.java:29)
```

The exception happened in the `substring` method of the `String` class, but the real culprit is the first method in a file that you wrote. In this example, that is `Homework2.main`, and you should look at line 29 of `Homework2.java`.

## HOW TO 6.1



### Working with Arrays

In many data processing situations, you need to process a sequence of values. This How To walks you through the steps for storing input values in an array and carrying out computations with the array elements.

**Problem Statement** Consider again the problem from Section 6.5: A final quiz score is computed by adding all the scores, except for the lowest one. For example, if the scores are

8 7 8.5 9.5 7 5 10

then the final score is 50.



Thierry Dosogne/The Image Bank/  
Getty Images, Inc.

#### Step 1 Decompose your task into steps.

You will usually want to break down your task into multiple steps, such as

- Reading the data into an array.
- Processing the data in one or more steps.
- Displaying the results.

When deciding how to process the data, you should be familiar with the array algorithms in Section 6.3. Most processing tasks can be solved by using one or more of these algorithms.

In our sample problem, we will want to read the data. Then we will remove the minimum and compute the total. For example, if the input is 8 7 8.5 9.5 7 5 10, we will remove the minimum of 5, yielding 8 7 8.5 9.5 7 10. The sum of those values is the final score of 50.

Thus, we have identified three steps:

**Read inputs.**  
**Remove the minimum.**  
**Calculate the sum.**

### Step 2

Determine which algorithm(s) you need.

Sometimes, a step corresponds to exactly one of the basic array algorithms in Section 6.3. That is the case with calculating the sum (Section 6.3.2) and reading the inputs (Section 6.3.10). At other times, you need to combine several algorithms. To remove the minimum value, you can find the minimum value (Section 6.3.3), find its position (Section 6.3.5), and remove the element at that position (Section 6.3.6).

We have now refined our plan as follows:

**Read inputs.**  
**Find the minimum.**  
**Find the position of the minimum.**  
**Remove the element at the position.**  
**Calculate the sum.**

This plan will work—see Section 6.5. But here is an alternate approach. It is easy to compute the sum and subtract the minimum. Then we don't have to find its position. The revised plan is

**Read inputs.**  
**Find the minimum.**  
**Calculate the sum.**  
**Subtract the minimum from the sum.**

### Step 3

Use methods to structure the program.

Even though it may be possible to put all steps into the `main` method, this is rarely a good idea. It is better to make each processing step into a separate method. In our example, we will implement three methods:

- `readInputs`
- `sum`
- `minimum`

The `main` method simply calls these methods:

```
double[] scores = readInputs();
double total = sum(scores) - minimum(scores);
System.out.println("Final score: " + total);
```

### Step 4

Assemble and test the program.

Place your methods into a class. Review your code and check that you handle both normal and exceptional situations. What happens with an empty array? One that contains a single element? When no match is found? When there are multiple matches? Consider these boundary conditions and make sure that your program works correctly.

In our example, it is impossible to compute the minimum if the array is empty. In that case, we should terminate the program with an error message *before* attempting to call the `minimum` method.

What if the minimum value occurs more than once? That means that a student had more than one test with the same low score. We subtract only one of the occurrences of that low score, and that is the desired behavior.

The following table shows test cases and their expected output:

| Test Case          | Expected Output | Comment                                               |
|--------------------|-----------------|-------------------------------------------------------|
| 8 7 8.5 9.5 7 5 10 | 50              | See Step 1.                                           |
| 8 7 7 9            | 24              | Only one instance of the low score should be removed. |
| 8                  | 0               | After removing the low score, no score remains.       |
| (no inputs)        | <b>Error</b>    | That is not a legal input.                            |

Here's the complete program (`how_to_1/Scores.java`):

```

import java.util.Arrays;
import java.util.Scanner;

/**
 * This program computes a final score for a series of quiz scores: the sum after dropping
 * the lowest score. The program uses arrays.
 */
public class Scores
{
 public static void main(String[] args)
 {
 double[] scores = readInputs();
 if (scores.length == 0)
 {
 System.out.println("At least one score is required.");
 }
 else
 {
 double total = sum(scores) - minimum(scores);
 System.out.println("Final score: " + total);
 }
 }

 /**
 * Reads a sequence of floating-point numbers.
 * @return an array containing the numbers
 */
 public static double[] readInputs()
 {
 // Read the input values into an array

 final int INITIAL_SIZE = 10;
 double[] inputs = new double[INITIAL_SIZE];
 System.out.println("Please enter values, Q to quit:");
 Scanner in = new Scanner(System.in);
 int currentSize = 0;
 while (in.hasNextDouble())
 {
 // Grow the array if it has been completely filled

```

```

 if (currentSize >= inputs.length)
 {
 inputs = Arrays.copyOf(inputs, 2 * inputs.length);
 }
 inputs[currentSize] = in.nextDouble();
 currentSize++;
 }

 return Arrays.copyOf(inputs, currentSize);
}

/**
 * Computes the sum of the values in an array.
 * @param values an array
 * @return the sum of the values in values
 */
public static double sum(double[] values)
{
 double total = 0;
 for (double element : values)
 {
 total = total + element;
 }
 return total;
}

/**
 * Gets the minimum value from an array.
 * @param values an array of size >= 1
 * @return the smallest element of values
 */
public static double minimum(double[] values)
{
 double smallest = values[0];
 for (int i = 1; i < values.length; i++)
 {
 if (values[i] < smallest)
 {
 smallest = values[i];
 }
 }
 return smallest;
}

```

---



### WORKED EXAMPLE 6.1

#### **Rolling the Dice**

This Worked Example shows how to analyze a set of die tosses to see whether the die is “fair”. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 6.1.



© ktsimage/iStockphoto.

## 6.6 Problem Solving: Discovering Algorithms by Manipulating Physical Objects

© JenCon/iStockphoto.



*Manipulating physical objects can give you ideas for discovering algorithms.*

Use a sequence of coins, playing cards, or toys to visualize an array of values.

In Section 6.5, you saw how to solve a problem by combining and adapting known algorithms. But what do you do when none of the standard algorithms is sufficient for your task? In this section, you will learn a technique for discovering algorithms by manipulating physical objects.

Consider the following task: You are given an array whose size is an even number, and you are to switch the first and the second half. For example, if the array contains the eight numbers

|   |    |    |   |    |   |   |   |
|---|----|----|---|----|---|---|---|
| 9 | 13 | 21 | 4 | 11 | 7 | 1 | 3 |
|---|----|----|---|----|---|---|---|

then you should change it to

|    |   |   |   |   |    |    |   |
|----|---|---|---|---|----|----|---|
| 11 | 7 | 1 | 3 | 9 | 13 | 21 | 4 |
|----|---|---|---|---|----|----|---|

Many students find it quite challenging to come up with an algorithm. They may know that a loop is required, and they may realize that elements should be inserted (Section 6.3.7) or swapped (Section 6.3.8), but they do not have sufficient intuition to draw diagrams, describe an algorithm, or write down pseudocode.

One useful technique for discovering an algorithm is to manipulate physical objects. Start by lining up some objects to denote an array. Coins, playing cards, or small toys are good choices.

Here we arrange eight coins:



Now let's step back and see what we can do to change the order of the coins. We can remove a coin (Section 6.3.6):



We can insert a coin (Section 6.3.7):



Or we can swap two coins (Section 6.3.8).

*Visualizing the swapping of two coins*



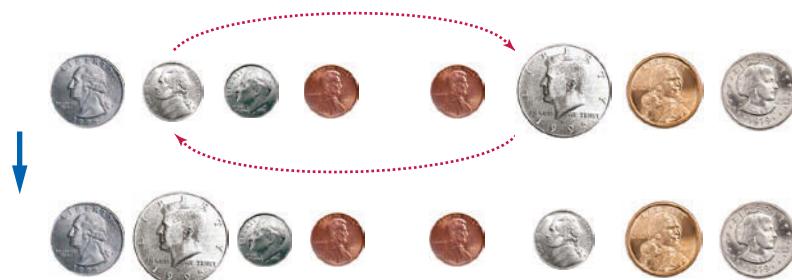
Go ahead—line up some coins and try out these three operations right now so that you get a feel for them.

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as Java programmers, we will say that we swap the coins in positions 0 and 4:



Next, we swap the coins in positions 1 and 5:



Two more swaps, and we are done:



Now an algorithm is becoming apparent:

```
i = 0
j = ... // We'll think about that in a minute
While // Don't know yet
 Swap elements at positions i and j.
 i++
 j++
```

Where does the variable *j* start? When we have eight coins, the coin at position zero is moved to position 4. In general, it is moved to the middle of the array, or to position *size / 2*.

And how many iterations do we make? We need to swap all coins in the first half. That is, we need to swap *size / 2* coins. The pseudocode is

```
i = 0
j = size / 2
While (i < size / 2)
 Swap elements at positions i and j.
 i++
 j++
```

It is a good idea to make a walkthrough of the pseudocode (see Section 4.2). You can use paper clips to denote the positions of the variables *i* and *j*. If the walkthrough is successful, then we know that there was no “off-by-one” error in the pseudocode. Self Check 29 asks you to carry out the walkthrough, and Exercise E6.8 asks you to translate the pseudocode to Java. Exercise R6.27 suggests a different algorithm for switching the two halves of an array, by repeatedly removing and inserting coins.

Many people find that the manipulation of physical objects is less intimidating than drawing diagrams or mentally envisioning algorithms. Give it a try when you need to design a new algorithm!

You can use paper clips as position markers or counters.

### SELF CHECK



29. Walk through the algorithm that we developed in this section, using two paper clips to indicate the positions for *i* and *j*. Explain why there are no bounds errors in the pseudocode.
30. Take out some coins and simulate the following pseudocode, using two paper clips to indicate the positions for *i* and *j*.

```
i = 0
j = size - 1
While i < j
 Swap elements at positions i and j.
 i++
 j--
```

What does the algorithm do?

31. Consider the task of rearranging all elements in an array so that the even numbers come first. Otherwise, the order doesn’t matter. For example, the array

1 4 14 2 1 3 5 6 23

could be rearranged to

4 2 14 6 1 5 3 23 1



### FULL CODE EXAMPLE

Go to [wiley.com/go/bj1o2code](http://wiley.com/go/bj1o2code) to download a program that implements the algorithm that switches the first and second halves of an array.

Using coins and paperclips, discover an algorithm that solves this task by swapping elements, then describe it in pseudocode.

32. Discover an algorithm for the task of Self Check 31 that uses removal and insertion of elements instead of swapping.
33. Consider the algorithm in Section 4.7.4 that finds the largest element in a sequence of inputs—*not* the largest element in an array. Why is this algorithm better visualized by picking playing cards from a deck rather than arranging toy soldiers in a sequence?



© claudioarnes/iStockphoto.

**Practice It** Now you can try these exercises at the end of the chapter: R6.27, R6.29, E6.8.



### VIDEO EXAMPLE 6.1

### Removing Duplicates from an Array

In this Video Example, we will discover an algorithm for removing duplicates from an array. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 6.1.

## 6.7 Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout. Such data sets commonly occur in financial and scientific applications. An arrangement consisting of rows and columns of values is called a *two-dimensional array*, or a *matrix*.

Let's explore how to store the example data shown in Figure 12: the medal counts of the figure skating competitions at the 2014 Winter Olympics.



© Trub/iStockphoto.



© technoir/iStockphoto.

|               | Gold | Silver | Bronze |
|---------------|------|--------|--------|
| Canada        | 0    | 3      | 0      |
| Italy         | 0    | 0      | 1      |
| Germany       | 0    | 0      | 1      |
| Japan         | 1    | 0      | 0      |
| Kazakhstan    | 0    | 0      | 1      |
| Russia        | 3    | 1      | 1      |
| South Korea   | 0    | 1      | 0      |
| United States | 1    | 0      | 1      |

**Figure 12** Figure Skating Medal Counts

### 6.7.1 Declaring Two-Dimensional Arrays

Use a two-dimensional array to store tabular data.

In Java, you obtain a two-dimensional array by supplying the number of rows and columns. For example, `new int[7][3]` is an array with seven rows and three columns. You store a reference to such an array in a variable of type `int[][]`. Here is a complete declaration of a two-dimensional array, suitable for holding our medal count data:

```
final int COUNTRIES = 8;
final int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

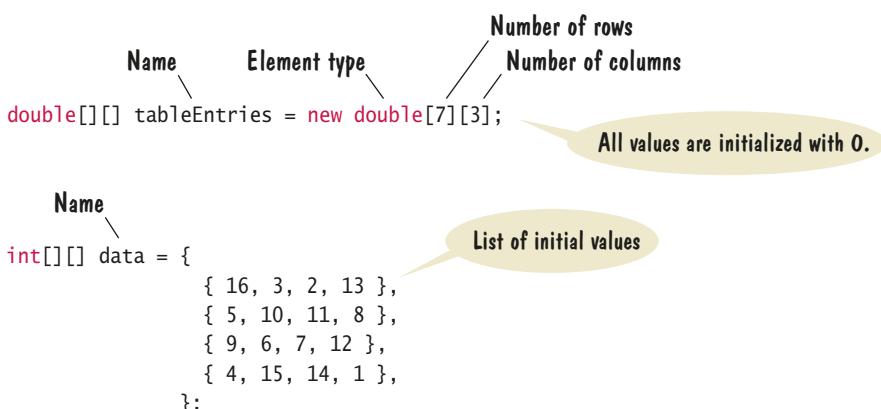
Alternatively, you can declare and initialize the array by grouping each row:

```
int[][] counts =
{
 { 0, 3, 0 },
 { 0, 0, 1 },
 { 0, 0, 1 },
 { 1, 0, 0 },
 { 0, 0, 1 },
 { 3, 1, 1 },
 { 0, 1, 0 }
};
```

As with one-dimensional arrays, you cannot change the size of a two-dimensional array once it has been declared.

### Syntax 6.3

#### Two-Dimensional Array Declaration



### 6.7.2 Accessing Elements

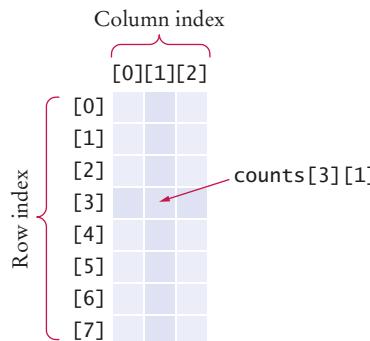
Individual elements in a two-dimensional array are accessed by using two index values, `array[i][j]`.

To access a particular element in the two-dimensional array, you need to specify two index values in separate brackets to select the row and column, respectively (see Figure 13):

```
int medalCount = counts[3][1];
```

To access all elements in a two-dimensional array, you use two nested loops. For example, the following loop prints all elements of counts:

```
for (int i = 0; i < COUNTRIES; i++)
{
 // Process the ith row
 for (int j = 0; j < MEDALS; j++)
 {
 // Process the jth column in the ith row
 System.out.printf("%8d", counts[i][j]);
 }
 System.out.println(); // Start a new line at the end of the row
}
```



**Figure 13**  
Accessing an Element in a  
Two-Dimensional Array

### 6.7.3 Locating Neighboring Elements

Some programs that work with two-dimensional arrays need to locate the elements that are adjacent to an element. This task is particularly common in games. Figure 14 shows how to compute the index values of the neighbors of an element.

For example, the neighbors of `counts[3][1]` to the left and right are `counts[3][0]` and `counts[3][2]`. The neighbors to the top and bottom are `counts[2][1]` and `counts[4][1]`.

You need to be careful about computing neighbors at the boundary of the array. For example, `counts[0][1]` has no neighbor to the top. Consider the task of computing the sum of the neighbors to the top and bottom of the element `count[i][j]`. You need to check whether the element is located at the top or bottom of the array:

```
int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

|                             |                         |                             |
|-----------------------------|-------------------------|-----------------------------|
| <code>[i - 1][j - 1]</code> | <code>[i - 1][j]</code> | <code>[i - 1][j + 1]</code> |
|                             |                         |                             |
| <code>[i][j - 1]</code>     | <code>[i][j]</code>     | <code>[i][j + 1]</code>     |

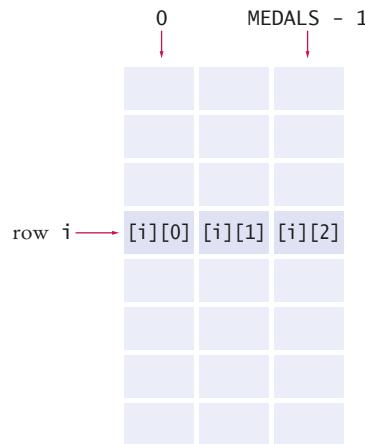
|                             |                         |                             |
|-----------------------------|-------------------------|-----------------------------|
| <code>[i + 1][j - 1]</code> | <code>[i + 1][j]</code> | <code>[i + 1][j + 1]</code> |
|                             |                         |                             |

**Figure 14**  
Neighboring Locations in a  
Two-Dimensional Array

### 6.7.4 Computing Row and Column Totals

A common task is to compute row or column totals. In our example, the row totals give us the total number of medals won by a particular country.

Finding the right index values is a bit tricky, and it is a good idea to make a quick sketch. To compute the total of row  $i$ , we need to visit the following elements:

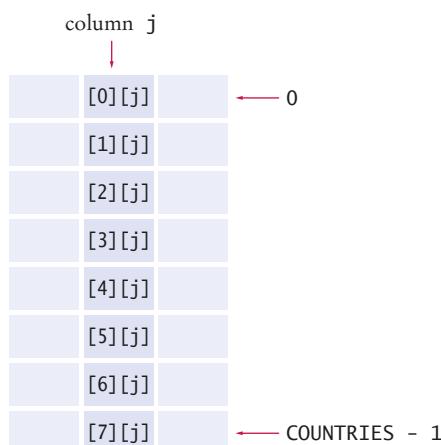


As you can see, we need to compute the sum of  $\text{counts}[i][j]$ , where  $j$  ranges from 0 to  $\text{MEDALS} - 1$ . The following loop computes the total:

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
 total = total + counts[i][j];
```

Computing column totals is similar. Form the sum of  $\text{counts}[i][j]$ , where  $i$  ranges from 0 to  $\text{COUNTRIES} - 1$ .

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
 total = total + counts[i][j];
```



## 6.7.5 Two-Dimensional Array Parameters

When you pass a two-dimensional array to a method, you will want to recover the dimensions of the array. If `values` is a two-dimensional array, then

- `values.length` is the number of rows.
- `values[0].length` is the number of columns. (See Special Topic 6.4 for an explanation of this expression.)

For example, the following method computes the sum of all elements in a two-dimensional array:

```
public static int sum(int[][] values)
{
 int total = 0;
 for (int i = 0; i < values.length; i++)
 {
 for (int j = 0; j < values[0].length; j++)
 {
 total = total + values[i][j];
 }
 }
 return total;
}
```

Working with two-dimensional arrays is illustrated in the following program. The program prints out the medal counts and the row totals.

### sec07/Medals.java

```
1 /**
2 * This program prints a table of medal winner counts with row totals.
3 */
4 public class Medals
5 {
6 public static void main(String[] args)
7 {
8 final int COUNTRIES = 8;
9 final int MEDALS = 3;
10
11 String[] countries =
12 {
13 "Canada",
14 "Italy",
15 "Germany",
16 "Japan",
17 "Kazakhstan",
18 "Russia",
19 "South Korea",
20 "United States"
21 };
22
23 int[][] counts =
24 {
25 { 0, 3, 0 },
26 { 0, 0, 1 },
27 { 0, 0, 1 },
28 { 1, 0, 0 },
29 { 0, 0, 1 },
30 { 0, 0, 1 },
31 { 0, 0, 1 },
32 { 0, 0, 1 }
33 }
34 }
```

```

30 { 3, 1, 1 },
31 { 0, 1, 0 },
32 { 1, 0, 1 }
33 };
34
35 System.out.println(" Country Gold Silver Bronze Total");
36
37 // Print countries, counts, and row totals
38 for (int i = 0; i < COUNTRIES; i++)
39 {
40 // Process the ith row
41 System.out.printf("%15s", countries[i]);
42
43 int total = 0;
44
45 // Print each row element and update the row total
46 for (int j = 0; j < MEDALS; j++)
47 {
48 System.out.printf("%8d", counts[i][j]);
49 total = total + counts[i][j];
50 }
51
52 // Display the row total and print a new line
53 System.out.printf("%8d%n", total);
54 }
55 }
56 }
```

### Program Run

| Country       | Gold | Silver | Bronze | Total |
|---------------|------|--------|--------|-------|
| Canada        | 0    | 3      | 0      | 3     |
| Italy         | 0    | 0      | 1      | 1     |
| Germany       | 0    | 0      | 1      | 1     |
| Japan         | 1    | 0      | 0      | 1     |
| Kazakhstan    | 0    | 0      | 1      | 1     |
| Russia        | 3    | 1      | 1      | 5     |
| South Korea   | 0    | 1      | 0      | 1     |
| United States | 1    | 0      | 1      | 2     |

### SELF CHECK



34. What results do you get if you total the columns in our sample data?

35. Consider an  $8 \times 8$  array for a board game:

```
int[][] board = new int[8][8];
```

Using two nested loops, initialize the board so that zeroes and ones alternate, as on a checkerboard:

```
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
.
.
1 0 1 0 1 0 1 0
```

*Hint:* Check whether  $i + j$  is even.

36. Declare a two-dimensional array for representing a tic-tac-toe board. The board has three rows and columns and contains strings "x", "o", and " ".

37. Write an assignment statement to place an "x" in the upper-right corner of the tic-tac-toe board in Self Check 36.
38. Which elements are on the diagonal joining the upper-left and the lower-right corners of the tic-tac-toe board in Self Check 36?

**Practice It** Now you can try these exercises at the end of the chapter: R6.29, P6.5, P6.6.



## WORKED EXAMPLE 6.2

### A World Population Table



This Worked Example shows how to print world population data in a table with row and column headers, and with totals for each of the data columns. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 6.2.

#### Special Topic 6.4



### Two-Dimensional Arrays with Variable Row Lengths

When you declare a two-dimensional array with the command

```
int[][] a = new int[3][3];
```

then you get a  $3 \times 3$  matrix that can store 9 elements:

```
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]
a[2][0] a[2][1] a[2][2]
```

In this matrix, all rows have the same length.

In Java it is possible to declare arrays in which the row length varies. For example, you can store an array that has a triangular shape, such as:

```
b[0][0]
b[1][0] b[1][1]
b[2][0] b[2][1] b[2][2]
```

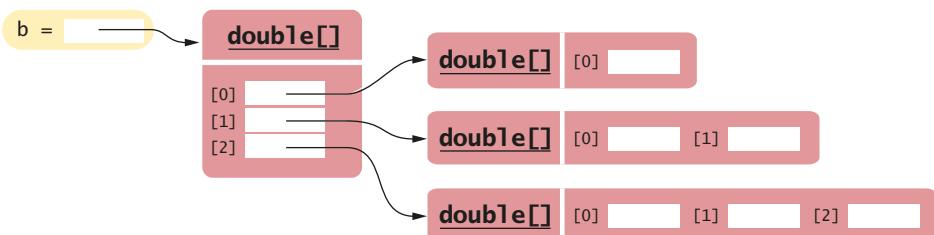
To allocate such an array, you must work harder. First, you allocate space to hold three rows. Indicate that you will manually set each row by leaving the second array index empty:

```
double[][] b = new double[3][];
```

Then allocate each row separately (see Figure 15):

```
for (int i = 0; i < b.length; i++)
{
 b[i] = new double[i + 1];
}
```

You can access each array element as `b[i][j]`. The expression `b[i]` selects the *i*th row, and the `[j]` operator selects the *j*th element in that row.



**Figure 15** A Triangular Array

Note that the number of rows is `b.length`, and the length of the *i*th row is `b[i].length`. For example, the following pair of loops prints a ragged array:

```
for (int i = 0; i < b.length; i++)
{
 for (int j = 0; j < b[i].length; j++)
 {
 System.out.print(b[i][j]);
 }
 System.out.println();
}
```

Alternatively, you can use two enhanced for loops:

```
for (double[] row : b)
{
 for (double element : row)
 {
 System.out.print(element);
 }
 System.out.println();
}
```

Naturally, such “ragged” arrays are not very common.

Java implements plain two-dimensional arrays in exactly the same way as ragged arrays: as arrays of one-dimensional arrays. The expression `new int[3][3]` automatically allocates an array of three rows, and three arrays for the rows’ contents.

### Special Topic 6.5



### Multidimensional Arrays

You can declare arrays with more than two dimensions. For example, here is a three-dimensional array:

```
int[][][] rubiksCube = new int[3][3][3];
```

Each array element is specified by three index values:

```
rubiksCube[i][j][k]
```

## 6.8 Array Lists

An array list stores a sequence of values whose size can change.

When you write a program that collects inputs, you don’t always know how many inputs you will have. In such a situation, an **array list** offers two significant advantages:

- Array lists can grow and shrink as needed.
- The `ArrayList` class supplies methods for common tasks, such as inserting and removing elements.

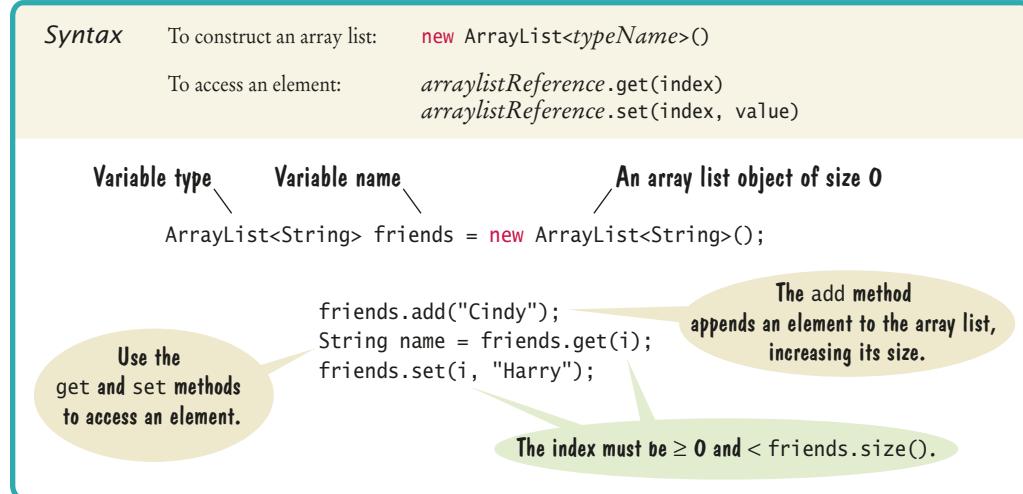
In the following sections, you will learn how to work with array lists.

*An array list expands to hold as many elements as needed.*



Michael Brake/Stockphoto.

## Syntax 6.4 Array Lists



### 6.8.1 Declaring and Using Array Lists

The following statement declares an array list of strings:

```
ArrayList<String> names = new ArrayList<String>();
```

The `ArrayList` class is a generic class:  
`ArrayList<Type>` collects elements of the specified type.

The `ArrayList` class is contained in the `java.util` package. In order to use array lists in your program, you need to use the statement `import java.util.ArrayList`.

The type `ArrayList<String>` denotes an array list of `String` elements. The angle brackets around the `String` type tell you that `String` is a **type parameter**. You can replace `String` with any other class and get a different array list type. For that reason, `ArrayList` is called a **generic class**. However, you cannot use primitive types as type parameters—there is no `ArrayList<int>` or `ArrayList<double>`. Section 6.8.5 shows how you can collect numbers in an array list.

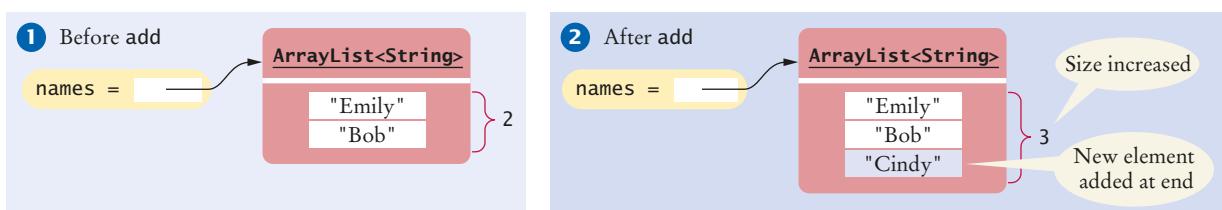
It is a common error to forget the initialization:

```
ArrayList<String> names;
names.add("Harry"); // Error—names not initialized
```

Here is the proper initialization:

```
ArrayList<String> names = new ArrayList<String>();
```

Note the `()` after `new ArrayList<String>` on the right-hand side of the initialization. It indicates that the **constructor** of the `ArrayList<String>` class is being called. We will discuss constructors in Chapter 8.



**Figure 16** Adding an Element with `add`

When the `ArrayList<String>` is first constructed, it has size 0. You use the `add` method to add an element to the end of the array list.

```
names.add("Emily"); // Now names has size 1 and element "Emily"
names.add("Bob"); // Now names has size 2 and elements "Emily", "Bob"
names.add("Cindy"); // names has size 3 and elements "Emily", "Bob", and "Cindy"
```

Use the `size` method to obtain the current size of an array list.

Use the `get` and `set` methods to access an array list element at a given index.

© Danijelj/iStockphoto.



An array list has methods for adding and removing elements in the middle.

The size increases after each call to add (see Figure 16). The `size` method yields the current size of the array list.

To obtain an array list element, use the `get` method, not the `[]` operator. As with arrays, index values start at 0. For example, `names.get(2)` retrieves the name with index 2, the third element in the array list:

```
String name = names.get(2);
```

As with arrays, it is an error to access a nonexistent element. A very common bounds error is to use the following:

```
int i = names.size();
name = names.get(i); // Error
```

The last valid index is `names.size() - 1`.

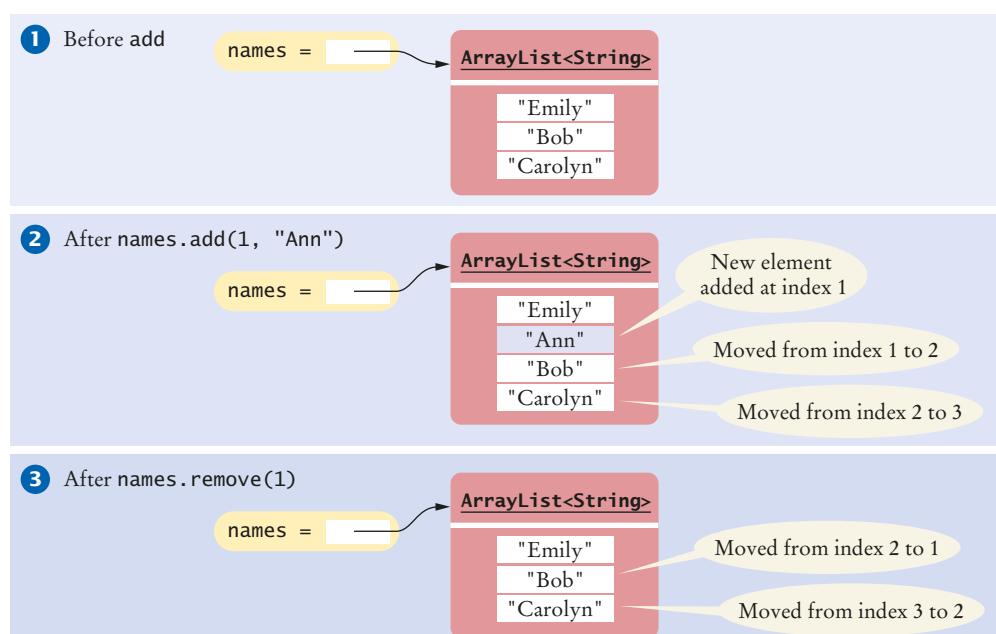
To set an array list element to a new value, use the `set` method.

```
names.set(2, "Carolyn");
```

This call sets position 2 of the `names` array list to "Carolyn", overwriting whatever value was there before.

The `set` method overwrites existing values. It is different from the `add` method, which adds a new element to the array list.

You can insert an element in the middle of an array list. For example, the call `names.add(1, "Ann")` adds a new element at position 1 and moves all elements with index 1 or larger by one position. After each call to the `add` method, the size of the array list increases by 1 (see Figure 17).



**Figure 17**  
Adding and  
Removing  
Elements in the  
Middle of an  
Array List

Use the add and remove methods to add and remove array list elements.

Conversely, the `remove` method removes the element at a given position, moves all elements after the removed element down by one position, and reduces the size of the array list by 1. Part 3 of Figure 17 illustrates the result of `names.remove(1)`.

With an array list, it is very easy to get a quick printout. Simply pass the array list to the `println` method:

```
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

## 6.8.2 Using the Enhanced for Loop with Array Lists

You can use the enhanced `for` loop to visit all elements of an array list. For example, the following loop prints all names:

```
ArrayList<String> names = . . . ;
for (String name : names)
{
 System.out.println(name);
}
```

This loop is equivalent to the following basic `for` loop:

```
for (int i = 0; i < names.size(); i++)
{
 String name = names.get(i);
 System.out.println(name);
}
```

**Table 2 Working with Array Lists**

|                                                                                                                                             |                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| <code>ArrayList&lt;String&gt; names = new ArrayList&lt;String&gt;();</code>                                                                 | Constructs an empty array list that can hold strings.                      |
| <code>names.add("Ann");</code><br><code>names.add("Cindy");</code>                                                                          | Adds elements to the end.                                                  |
| <code>System.out.println(names);</code>                                                                                                     | Prints [Ann, Cindy].                                                       |
| <code>names.add(1, "Bob");</code>                                                                                                           | Inserts an element at index 1.<br>names is now [Ann, Bob, Cindy].          |
| <code>names.remove(0);</code>                                                                                                               | Removes the element at index 0.<br>names is now [Bob, Cindy].              |
| <code>names.set(0, "Bill");</code>                                                                                                          | Replaces an element with a different value.<br>names is now [Bill, Cindy]. |
| <code>String name = names.get(i);</code>                                                                                                    | Gets an element.                                                           |
| <code>String last = names.get(names.size() - 1);</code>                                                                                     | Gets the last element.                                                     |
| <code>ArrayList&lt;Integer&gt; squares = new ArrayList&lt;Integer&gt;(); for (int i = 0; i &lt; 10; i++) {     squares.add(i * i); }</code> | Constructs an array list holding the first ten squares.                    |

### 6.8.3 Copying Array Lists

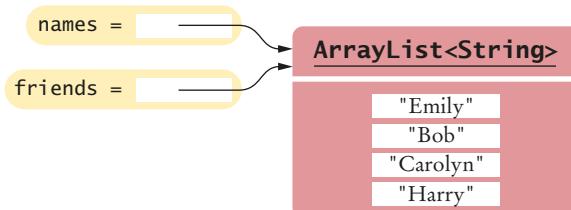
As with arrays, you need to remember that array list variables hold references. Copying the reference yields two references to the same array list (see Figure 18).

```
ArrayList<String> friends = names;
friends.add("Harry");
```

Now both `names` and `friends` reference the same array list to which the string "Harry" was added.

If you want to make a copy of an array list, construct the copy and pass the original list into the constructor:

```
ArrayList<String> newNames = new ArrayList<String>(names);
```



**Figure 18** Copying an ArrayList Reference

### 6.8.4 Array Lists and Methods

Like arrays, array lists can be method arguments and return values. Here is an example: a method that receives a list of strings and returns the reversed list.

```
public static ArrayList<String> reverse(ArrayList<String> names)
{
 // Allocate a list to hold the method result
 ArrayList<String> result = new ArrayList<String>();

 // Traverse the names list in reverse order, starting with the last element
 for (int i = names.size() - 1; i >= 0; i--)
 {
 // Add each name to the result
 result.add(names.get(i));
 }
 return result;
}
```

If this method is called with an array list containing the names Emily, Bob, Cindy, it returns a new array list with the names Cindy, Bob, Emily.

### 6.8.5 Wrappers and Auto-boxing

To collect numbers in array lists, you must use wrapper classes.

In Java, you cannot directly insert primitive type values—numbers, characters, or boolean values—into array lists. For example, you cannot form an `ArrayList<double>`. Instead, you must use one of the **wrapper classes** shown in the following table.

| Primitive Type | Wrapper Class |
|----------------|---------------|
| byte           | Byte          |
| boolean        | Boolean       |
| char           | Character     |
| double         | Double        |
| float          | Float         |
| int            | Integer       |
| long           | Long          |
| short          | Short         |

For example, to collect `double` values in an `ArrayList<Double>`. Note that the wrapper class names start with uppercase letters, and that two of them differ from the names of the corresponding primitive type: `Integer` and `Character`.

Conversion between primitive types and the corresponding wrapper classes is automatic. This process is called **auto-boxing** (even though *auto-wrapping* would have been more consistent).

For example, if you assign a `double` value to a `Double` variable, the number is automatically “put into a box” (see Figure 19).

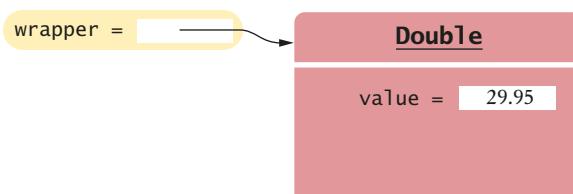
```
Double wrapper = 29.95;
```

Conversely, wrapper values are automatically “unboxed” to primitive types.

```
double x = wrapper;
```

Because boxing and unboxing is automatic, you don’t need to think about it. Simply remember to use the wrapper type when you declare array lists of numbers. From then on, use the primitive type and rely on auto-boxing.

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```



**Figure 19** A Wrapper Class Variable

*Like truffles that must be in a wrapper to be sold, a number must be placed in a wrapper to be stored in an array list.*

## 6.8.6 Using Array Algorithms with Array Lists

The array algorithms in Section 6.3 can be converted to array lists simply by using the array list methods instead of the array syntax (see Table 3 on page 309). For example, this code snippet finds the largest element in an array:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
 if (values[i] > largest)
 {
 largest = values[i];
 }
}
```

Here is the same algorithm, now using an array list:

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
 if (values.get(i) > largest)
 {
 largest = values.get(i);
 }
}
```

## 6.8.7 Storing Input Values in an Array List

When you collect an unknown number of inputs, array lists are *much* easier to use than arrays. Simply read inputs and add them to an array list:

```
ArrayList<Double> inputs = new ArrayList<Double>();
while (in.hasNextDouble())
{
 inputs.add(in.nextDouble());
}
```

## 6.8.8 Removing Matches

It is easy to remove elements from an array list, by calling the `remove` method. A common processing task is to remove all elements that match a particular condition. Suppose, for example, that we want to remove all strings of length  $< 4$  from an array list.

Of course, you traverse the array list and look for matching elements:

```
ArrayList<String> words = ...;
for (int i = 0; i < words.size(); i++)
{
 String word = words.get(i);
 if (word.length() < 4)
 {
 Remove the element at index i.
 }
}
```

But there is a subtle problem. After you remove the element, the `for` loop increments `i`, skipping past the *next* element.

Consider this concrete example, where `words` contains the strings "Welcome", "to", "the", "island!". When `i` is 1, we remove the word "to" at index 1. Then `i` is incremented to 2, and the word "the", which is now at position 1, is never examined.

| <code>i</code> | <code>words</code>               |
|----------------|----------------------------------|
| 0              | "Welcome", "to", "the", "island" |
| 1              | "Welcome", "the", "island"       |
| 2              |                                  |

We should not increment the index when removing a word. The appropriate pseudo-code is

```
If the element at index i matches the condition
 Remove the element at index i.
Else
 Increment i.
```

Because we don't always increment the index, a `for` loop is not appropriate for this algorithm. Instead, use a `while` loop:

```
int i = 0;
while (i < words.size())
{
 String word = words.get(i);
 if (word.length() < 4)
 {
 words.remove(i);
 }
 else
 {
 i++;
 }
}
```

### 6.8.9 Choosing Between Array Lists and Arrays

For most programming tasks, array lists are easier to use than arrays. Array lists can grow and shrink. On the other hand, arrays have a nicer syntax for element access and initialization.

Which of the two should you choose? Here are some recommendations.

- If the size of a collection never changes, use an array.
- If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
- Otherwise, use an array list.

The following program shows how to mark the largest value in a sequence of values. This program uses an array list. Note how the program is an improvement over the array version on page 277. This program can process input sequences of arbitrary length.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a version of the Scores program that uses an array list.

**Table 3 Comparing Array and Array List Operations**

| Operation                                  | Arrays                                                                 | Array Lists                                          |
|--------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------|
| Get an element.                            | <code>x = values[4];</code>                                            | <code>x = values.get(4);</code>                      |
| Replace an element.                        | <code>values[4] = 35;</code>                                           | <code>values.set(4, 35);</code>                      |
| Number of elements.                        | <code>values.length</code>                                             | <code>values.size()</code>                           |
| Number of filled elements.                 | <code>currentSize</code><br>(companion variable, see<br>Section 6.1.3) | <code>values.size()</code>                           |
| Remove an element.                         | See Section 6.3.6.                                                     | <code>values.remove(4);</code>                       |
| Add an element, growing<br>the collection. | See Section 6.3.7.                                                     | <code>values.add(35);</code>                         |
| Initializing a collection.                 | <code>int[] values = { 1, 4, 9 };</code>                               | No initializer list syntax;<br>call add three times. |

**sec08/LargestInArrayList.java**

```

1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 /**
5 This program reads a sequence of values and prints them, marking the largest value.
6 */
7 public class LargestInArrayList
8 {
9 public static void main(String[] args)
10 {
11 ArrayList<Double> values = new ArrayList<Double>();
12
13 // Read inputs
14
15 System.out.println("Please enter values, Q to quit:");
16 Scanner in = new Scanner(System.in);
17 while (in.hasNextDouble())
18 {
19 values.add(in.nextDouble());
20 }
21
22 // Find the largest value
23
24 double largest = values.get(0);
25 for (int i = 1; i < values.size(); i++)
26 {
27 if (values.get(i) > largest)
28 {
29 largest = values.get(i);
30 }
31 }
32
33 // Print all values, marking the largest
34

```

```

35 for (double element : values)
36 {
37 System.out.print(element);
38 if (element == largest)
39 {
40 System.out.print(" <= largest value");
41 }
42 System.out.println();
43 }
44 }
45 }
```

**Program Run**

Please enter values, Q to quit:  
 35 80 115 44.5 Q  
 35  
 80  
 115 <= largest value  
 44.5

**SELF CHECK**

39. Declare an array list `primes` of integers that contains the first five prime numbers (2, 3, 5, 7, and 11).
40. Given the array list `primes` declared in Self Check 39, write a loop to print its elements in reverse order, starting with the last element.
41. What does the array list `names` contain after the following statements?

```
ArrayList<String> names = new ArrayList<String>;
names.add("Bob");
names.add(0, "Ann");
names.remove(1);
names.add("Cal");
```

42. What is wrong with this code snippet?

```
ArrayList<String> names;
names.add(Bob);
```

43. Consider this method that appends the elements of one array list to another.

```
public static void append(ArrayList<String> target, ArrayList<String> source)
{
 for (int i = 0; i < source.size(); i++)
 {
 target.add(source.get(i));
 }
}
```

What are the contents of `names1` and `names2` after these statements?

```
ArrayList<String> names1 = new ArrayList<String>();
names1.add("Emily");
names1.add("Bob");
names1.add("Cindy");
ArrayList<String> names2 = new ArrayList<String>();
names2.add("Dave");
append(names1, names2);
```

44. Suppose you want to store the names of the weekdays. Should you use an array list or an array of seven strings?

- 45.** The sec08 directory of your source code contains an alternate implementation of the problem solution in How To 6.1 on page 287. Compare the array and array list implementations. What is the primary advantage of the latter?

**Practice It** Now you can try these exercises at the end of the chapter: R6.11, R6.33, E6.15, E6.18.

### Common Error 6.4



#### Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not at all consistent.

| Data Type  | Number of Elements |
|------------|--------------------|
| Array      | a.length           |
| Array list | a.size()           |
| String     | a.length()         |

It is a common error to confuse these. You just have to remember the correct syntax for every data type.

### Special Topic 6.6



#### The Diamond Syntax

There is a convenient syntax enhancement for declaring array lists and other generic classes. In a statement that declares and constructs an array list, you need not repeat the type parameter in the constructor. That is, you can write

```
ArrayList<String> names = new ArrayList<>();
```

instead of

```
ArrayList<String> names = new ArrayList<String>();
```

This shortcut is called the “diamond syntax” because the empty brackets <> look like a diamond shape.

For now, we will use the explicit syntax and include the type parameters with constructors. In later chapters, we will switch to the diamond syntax.



#### VIDEO EXAMPLE 6.2

#### Game of Life

Conway's *Game of Life* simulates the growth of a population, using only two simple rules. This Video Example shows you how to implement this famous “game”. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 6.2.



## CHAPTER SUMMARY

### Use arrays for collecting values.



- An array collects a sequence of values of the same type.
- Individual elements in an array are accessed by an integer index  $i$ , using the notation  $\text{array}[i]$ .
- An array element can be used like any variable.
- An array index must be at least zero and less than the size of the array.
- A bounds error, which occurs if you supply an invalid array index, can cause your program to terminate.
- Use the expression  $\text{array.length}$  to find the number of elements in an array.
- An array reference specifies the location of an array. Copying the reference yields a second reference to the same array.
- With a partially filled array, keep a companion variable for the current size.



### Know when to use the enhanced for loop.

- You can use the enhanced for loop to visit all elements of an array.
- Use the enhanced for loop if you do not need the index values in the loop body.

### Know and use common array algorithms.



- When separating elements, don't place a separator before the first element.
- A linear search inspects elements in sequence until a match is found.
- Before inserting an element, move elements to the end of the array *starting with the last one*.
- Use a temporary variable when swapping two elements.
- Use the `Arrays.copyOf` method to copy the elements of an array into a new array.

### Implement methods that process arrays.

- Arrays can occur as method arguments and return values.

### Combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

### Discover algorithms by manipulating physical objects.



- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

**Use two-dimensional arrays for data that is arranged in rows and columns.**

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two index values, `array[i][j]`.

**Use array lists for managing collections whose size can change.**

- An array list stores a sequence of values whose size can change.
- The `ArrayList` class is a generic class: `ArrayList<Type>` collects elements of the specified type.
- Use the `size` method to obtain the current size of an array list.
- Use the `get` and `set` methods to access an array list element at a given index.
- Use the `add` and `remove` methods to add and remove array list elements.
- To collect numbers in array lists, you must use wrapper classes.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

`java.lang.Boolean`  
`java.lang.Double`  
`java.lang.Integer`  
`java.util.Arrays`  
`copyOf`  
`toString`

`java.util.ArrayList<E>`  
`add`  
`get`  
`remove`  
`set`  
`size`

**REVIEW EXERCISES****R6.1** Carry out the following tasks with an array:

- Allocate an array `a` of ten integers.
- Put the number 17 as the initial element of the array.
- Put the number 29 as the last element of the array.
- Fill the remaining elements with `-1`.
- Add 1 to each element of the array.
- Print all elements of the array, one per line.
- Print all elements of the array in a single line, separated by commas.

**■■ R6.2** Write code that fills an array values with each set of numbers below.

- a. 1 2 3 4 5 6 7 8 9 10
- b. 0 2 4 6 8 10 12 14 16 18 20
- c. 1 4 9 16 25 36 49 64 81 100
- d. 0 0 0 0 0 0 0 0 0 0
- e. 1 4 9 16 9 7 4 9 11
- f. 0 1 0 1 0 1 0 1 0 1
- g. 0 1 2 3 4 0 1 2 3 4

**■■ R6.3** Consider the following array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What is the value of total after the following loops complete?

- a. int total = 0;  
for (int i = 0; i < 10; i++) { total = total + a[i]; }
- b. int total = 0;  
for (int i = 0; i < 10; i = i + 2) { total = total + a[i]; }
- c. int total = 0;  
for (int i = 1; i < 10; i = i + 2) { total = total + a[i]; }
- d. int total = 0;  
for (int i = 2; i <= 10; i++) { total = total + a[i]; }
- e. int total = 0;  
for (int i = 1; i < 10; i = 2 \* i) { total = total + a[i]; }
- f. int total = 0;  
for (int i = 9; i >= 0; i--) { total = total + a[i]; }
- g. int total = 0;  
for (int i = 9; i >= 0; i = i - 2) { total = total + a[i]; }
- h. int total = 0;  
for (int i = 0; i < 10; i++) { total = a[i] - total; }

**■■ R6.4** Consider the following array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What are the contents of the array a after the following loops complete?

- a. for (int i = 1; i < 10; i++) { a[i] = a[i - 1]; }
- b. for (int i = 9; i > 0; i--) { a[i] = a[i - 1]; }
- c. for (int i = 0; i < 9; i++) { a[i] = a[i + 1]; }
- d. for (int i = 8; i >= 0; i--) { a[i] = a[i + 1]; }
- e. for (int i = 1; i < 10; i++) { a[i] = a[i] + a[i - 1]; }
- f. for (int i = 1; i < 10; i = i + 2) { a[i] = 0; }
- g. for (int i = 0; i < 5; i++) { a[i + 5] = a[i]; }
- h. for (int i = 1; i < 5; i++) { a[i] = a[9 - i]; }

**■■■ R6.5** Write a loop that fills an array values with ten random numbers between 1 and 100. Write code for two nested loops that fill values with ten *different* random numbers between 1 and 100.

**■■■ R6.6** Write Java code for a loop that simultaneously computes both the maximum and minimum of an array.

■ **R6.7** What is wrong with each of the following code segments?

- a. 

```
int[] values = new int[10];
for (int i = 1; i <= 10; i++)
{
 values[i] = i * i;
}
```
- b. 

```
int[] values;
for (int i = 0; i < values.length; i++)
{
 values[i] = i * i;
}
```

■ ■ **R6.8** Write enhanced for loops for the following tasks.

- a. Printing all elements of an array in a single row, separated by spaces.
- b. Computing the product of all elements in an array.
- c. Counting how many elements in an array are negative.

■ ■ **R6.9** Rewrite the following loops without using the enhanced for loop construct. Here, `values` is an array of floating-point numbers.

- a. 

```
for (double x : values) { total = total + x; }
```
- b. 

```
for (double x : values) { if (x == target) { return true; } }
```
- c. 

```
int i = 0;
for (double x : values) { values[i] = 2 * x; i++; }
```

■ ■ **R6.10** Rewrite the following loops, using the enhanced for loop construct. Here, `values` is an array of floating-point numbers.

- a. 

```
for (int i = 0; i < values.length; i++) { total = total + values[i]; }
```
- b. 

```
for (int i = 1; i < values.length; i++) { total = total + values[i]; }
```
- c. 

```
for (int i = 0; i < values.length; i++)
{
 if (values[i] == target) { return i; }
}
```

■ **R6.11** What is wrong with each of the following code segments?

- a. 

```
ArrayList<int> values = new ArrayList<int>();
```
- b. 

```
ArrayList<Integer> values = new ArrayList<int>();
```
- c. 

```
ArrayList<Integer> values = new ArrayList<Integer>;
```
- d. 

```
ArrayList<Integer> values = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++)
{
 values.set(i - 1, i * i);
}
```
- e. 

```
ArrayList<Integer> values;
for (int i = 1; i <= 10; i++)
{
 values.add(i * i);
}
```

■ **R6.12** What is an index of an array? What are the legal index values? What is a bounds error?

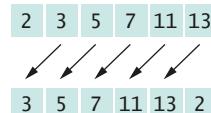
- R6.13 Write a program that contains a bounds error. Run the program. What happens on your computer?
- R6.14 Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.
- R6.15 Trace the flow of the linear search loop in Section 6.3.5, where `values` contains the elements 80 90 100 120 110. Show two columns, for `pos` and `found`. Repeat the trace when `values` contains 80 90 100 70.
- R6.16 Trace both mechanisms for removing an element described in Section 6.3.6. Use an array `values` with elements 110 90 100 120 80, and remove the element at index 2.
- R6.17 For the operations on partially filled arrays below, provide the header of a method. Do not implement the methods.
  - a. Sort the elements in decreasing order.
  - b. Print all elements, separated by a given string.
  - c. Count how many elements are less than a given value.
  - d. Remove all elements that are less than a given value.
  - e. Place all elements that are less than a given value in another array.

- R6.18 Trace the flow of the loop in Section 6.3.4 with the given example. Show two columns, one with the value of `i` and one with the output.
- R6.19 Consider the following loop for collecting all elements that match a condition; in this case, that the element is larger than 100.

```
ArrayList<Double> matches = new ArrayList<Double>();
for (double element : values)
{
 if (element > 100)
 {
 matches.add(element);
 }
}
```

Trace the flow of the loop, where `values` contains the elements 110 90 100 120 80. Show two columns, for `element` and `matches`.

- R6.20 Give pseudocode for an algorithm that rotates the elements of an array by one position, moving the initial element to the end of the array, like this:



- R6.21 Give pseudocode for an algorithm that removes all negative values from an array, preserving the order of the remaining elements.
- R6.22 Suppose `values` is a *sorted* array of integers. Give pseudocode that describes how a new value can be inserted in its proper position so that the resulting array stays sorted.

- R6.23** A *run* is a sequence of adjacent repeated values. Give pseudocode for computing the length of the longest run in an array. For example, the longest run in the array with elements

1 2 5 5 3 1 2 4 3 2 2 2 2 3 6 5 5 6 3 1

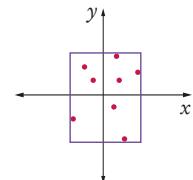
has length 4.

- R6.24** What is wrong with the following method that aims to fill an array with random numbers?

```
public static void fillWithRandomNumbers(double[] values)
{
 double[] numbers = new double[values.length];
 for (int i = 0; i < numbers.length; i++)
 {
 numbers[i] = Math.random();
 }
 values = numbers;
}
```

- R6.25** You are given two arrays denoting  $x$ - and  $y$ -coordinates of a set of points in the plane. For plotting the point set, we need to know the  $x$ - and  $y$ -coordinates of the smallest rectangle containing the points.

How can you obtain these values from the fundamental algorithms in Section 6.3?



- R6.26** Solve the problem described in Section 6.5 by sorting the array first. How do you need to modify the algorithm for computing the total?

- R6.27** Solve the task described in Section 6.6 using an algorithm that removes and inserts elements instead of switching them. Write the pseudocode for the algorithm, assuming that methods for removal and insertion exist. Act out the algorithm with a sequence of coins and explain why it is less efficient than the swapping algorithm developed in Section 6.6.

- R6.28** Develop an algorithm for finding the most frequently occurring value in an array of numbers. Use a sequence of coins. Place paper clips below each coin that count how many other coins of the same value are in the sequence. Give the pseudocode for an algorithm that yields the correct answer, and describe how using the coins and paper clips helped you find the algorithm.

- R6.29** Write Java statements for performing the following tasks with an array declared as

```
int[][] values = new int[ROWS][COLUMNS];
```

- Fill all entries with 0.
- Fill elements alternately with 0s and 1s in a checkerboard pattern.
- Fill only the elements at the top and bottom row with zeroes.
- Compute the sum of all elements.
- Print the array in tabular form.

- R6.30** Write pseudocode for an algorithm that fills the first and last column as well as the first and last row of a two-dimensional array of integers with -1.

- R6.31** Section 6.8.8 shows that you must be careful about updating the index value when you remove elements from an array list. Show how you can avoid this problem by traversing the array list backwards.
- R6.32** True or false?
- All elements of an array are of the same type.
  - Arrays cannot contain strings as elements.
  - Two-dimensional arrays always have the same number of rows and columns.
  - Elements of different columns in a two-dimensional array can have different types.
  - A method cannot return a two-dimensional array.
  - A method cannot change the length of an array argument.
  - A method cannot change the number of columns of an argument that is a two-dimensional array.
- R6.33** How do you perform the following tasks with array lists in Java?
- Test that two array lists contain the same elements in the same order.
  - Copy one array list to another.
  - Fill an array list with zeroes, overwriting all elements in it.
  - Remove all elements from an array list.
- R6.34** True or false?
- All elements of an array list are of the same type.
  - Array list index values must be integers.
  - Array lists cannot contain strings as elements.
  - Array lists can change their size, getting larger or smaller.
  - A method cannot return an array list.
  - A method cannot change the size of an array list argument.

### PRACTICE EXERCISES

- E6.1** Write a program that initializes an array with ten random integers and then prints four lines of output, containing
- Every element at an even index.
  - Every even element.
  - All elements in reverse order.
  - Only the first and last element.
- E6.2** Write array methods that carry out the following tasks for an array of integers. For each method, provide a test program.
- Swap the first and last elements in the array.
  - Shift all elements by one to the right and move the last element into the first position. For example, 1 4 9 16 25 would be transformed into 25 1 4 9 16.
  - Replace all even elements with 0.
  - Replace each element except the first and last by the larger of its two neighbors.

- e. Remove the middle element if the array length is odd, or the middle two elements if the length is even.
- f. Move all even elements to the front, otherwise preserving the order of the elements. Return true if all elements of the array are identical.
- g. Return the second-largest element in the array.
- h. Return true if the array is currently sorted in increasing order.
- i. Return true if the array contains two adjacent duplicate elements.
- j. Return true if the array contains duplicate elements (which need not be adjacent).

- E6.3 Modify the `LargestInArray.java` program in Section 6.3 to mark both the smallest and the largest elements.
- E6.4 Write a method `sumWithoutSmallest` that computes the sum of an array of values, except for the smallest one, in a single loop. In the loop, update the sum and the smallest value. After the loop, return the difference.
- E6.5 Write a method `public static void removeMin` that removes the minimum value from a partially filled array without calling other methods.
- E6.6 Compute the *alternating sum* of all elements in an array. For example, if your program reads the input

1 4 9 16 9 7 4 9 11

then it computes

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

- E6.7 Write a method that reverses the sequence of elements in an array. For example, if you call the method with the array

1 4 9 16 9 7 4 9 11

then the array is changed to

11 9 4 7 9 16 9 4 1

- E6.8 Write a method that implements the algorithm developed in Section 6.6.

- E6.9 Write a method

```
public static boolean equals(int[] a, int[] b)
```

that checks whether two arrays have the same elements in the same order.

- E6.10 Write a method

```
public static boolean sameSet(int[] a, int[] b)
```

that checks whether two arrays have the same elements in some order, ignoring duplicates. For example, the two arrays

1 4 9 16 9 7 4 9 11

and

11 11 7 9 16 4 1

would be considered identical. You will probably need one or more helper methods.

**•• E6.11** Write a method

```
public static boolean sameElements(int[] a, int[] b)
```

that checks whether two arrays have the same elements in some order, with the same multiplicities. For example,

1 4 9 16 9 7 4 9 11

and

11 1 4 9 16 9 7 4 9

would be considered identical, but

1 4 9 16 9 7 4 9 11

and

11 11 7 9 16 4 1 4 9

would not. You will probably need one or more helper methods.

**•• E6.12** Write a program that generates a sequence of 20 random values between 0 and 99 in an array, prints the sequence, sorts it, and prints the sorted sequence. Use the `sort` method from the standard Java library.**•• E6.13** Write a program that produces ten random permutations of the numbers 1 to 10. To generate a random permutation, you need to fill an array with the numbers 1 to 10 so that no two entries of the array have the same contents. You could do it by brute force, by generating random values until you have a value that is not yet in the array. But that is inefficient. Instead, follow this algorithm.

**Make a second array and fill it with the numbers 1 to 10.**

**Repeat 10 times**

**Pick a random position in the second array.**

**Remove the element at the position from the second array.**

**Append the removed element to the permutation array.**

**•• E6.14** Write a method that computes the average of the neighbors of a two-dimensional array element in the eight directions shown in Figure 14.

```
public static double neighborAverage(int[][] values, int row, int column)
```

However, if the element is located at the boundary of the array, only include the neighbors that are in the array. For example, if `row` and `column` are both 0, there are only three neighbors.

**•• E6.15** Write a program that reads a sequence of input values and displays a bar chart of the values, using asterisks, like this:

```



```

You may assume that all values are positive. First figure out the maximum value. That value's bar should be drawn with 40 asterisks. Shorter bars should use proportionally fewer asterisks.

- E6.16** Repeat Exercise E6.15, but make the bars vertical, with the tallest bar twenty asterisks high.

```
*
*
*
*
*
*
**


```

- E6.17** Improve the program of Exercise E6.15 to work correctly when the data set contains negative values.

- E6.18** Improve the program of Exercise E6.15 by adding captions for each bar. Prompt the user for the captions and data values. The output should look like this:

|             |       |
|-------------|-------|
| Egypt       | ***** |
| France      | ***** |
| Japan       | ***** |
| Uruguay     | ***** |
| Switzerland | ***** |

- E6.19** Write a method

`public static ArrayList<Integer> append(ArrayList<Integer> a, ArrayList<Integer> b)`  
that appends one array list after another. For example, if a is

1 4 9 16

and b is

9 7 4 9 11

then append returns the array list

1 4 9 16 9 7 4 9 11

- E6.20** Write a method

`public static ArrayList<Integer> merge(ArrayList<Integer> a, ArrayList<Integer> b)`  
that merges two array lists, alternating elements from both array lists. If one array list is shorter than the other, then alternate as long as you can and then append the remaining elements from the longer array list. For example, if a is

1 4 9 16

and b is

9 7 4 9 11

then merge returns the array list

1 9 4 7 9 4 16 9 11

**■■ E6.21** Write a method

```
public static ArrayList<Integer> mergeSorted(ArrayList<Integer> a,
 ArrayList<Integer> b)
```

that merges two *sorted* array lists, producing a new sorted array list. Keep an index into each array list, indicating how much of it has been processed already. Each time, append the smallest unprocessed element from either array list, then advance the index. For example, if a is

1 4 9 16

and b is

4 7 9 9 11

then `mergeSorted` returns the array list

1 4 4 7 9 9 9 11 16

- E6.22** Write a method that modifies an `ArrayList<String>`, moving all strings starting with an uppercase letter to the front, without otherwise changing the order of the elements.
- E6.23** Write a method that counts the number of distinct elements in an `ArrayList<String>`. Do not modify the array list.
- E6.24** Write a method that finds the first occurrence of a value in a two-dimensional array. Return an `int[]` array of length 2 with the row and column, or `null` if the value was not found.
- E6.25** Write a method that checks whether all elements in a two-dimensional array are identical.
- E6.26** Write a method that checks whether all elements in a two-dimensional array are distinct.
- E6.27** Write a method that checks whether two two-dimensional arrays are equal; that is, whether they have the same number of rows and columns, and corresponding elements are equal.
- E6.28** Write a method that swaps two rows of a two-dimensional array.
- E6.29** Write a method that swaps two columns of a two-dimensional array.

### PROGRAMMING PROJECTS

- P6.1** A *run* is a sequence of adjacent repeated values. Write a program that generates a sequence of 20 random die tosses in an array and that prints the die values, marking the runs by including them in parentheses, like this:

1 2 (5 5) 3 1 2 4 3 (2 2 2 2) 3 6 (5 5) 6 3 1

Use the following pseudocode:

```
inRun = false
For each valid index i in the array
 If inRun
 If values[i] is different from the preceding value
 Print).
 inRun = false
```

```

If not inRun
 If values[i] is the same as the following value
 Print (. .
 inRun = true
 Print values[i].
 If inRun, print).

```

- P6.2 Write a program that generates a sequence of 20 random die tosses in an array and that prints the die values, marking only the longest run, like this:

1 2 5 5 3 1 2 4 3 (2 2 2 2) 3 6 5 5 6 3 1

If there is more than one run of maximum length, mark the first one.

- P6.3 It is a well-researched fact that men in a restroom generally prefer to maximize their distance from already occupied stalls, by occupying the middle of the longest sequence of unoccupied places.

For example, consider the situation where ten stalls are empty.

-----

The first visitor will occupy a middle position:

----- X -----

The next visitor will be in the middle of the empty area at the left.

--- X --- X -----

Write a program that reads the number of stalls and then prints out diagrams in the format given above when the stalls become filled, one at a time. Hint: Use an array of boolean values to indicate whether a stall is occupied.

- P6.4 In this assignment, you will model the game of *Bulgarian Solitaire*. The game starts with 45 cards. (They need not be playing cards. Unmarked index cards work just as well.) Randomly divide them into some number of piles of random size. For example, you might start with piles of size 20, 5, 1, 9, and 10. In each round, you take one card from each pile, forming a new pile with these cards. For example, the sample starting configuration would be transformed into piles of size 19, 4, 8, 9, and 5. The solitaire is over when the piles have size 1, 2, 3, 4, 5, 6, 7, 8, and 9, in some order. (It can be shown that you always end up with such a configuration.)

In your program, produce a random starting configuration and print it. Then keep applying the solitaire step and print the result. Stop when the solitaire final configuration is reached.

- P6.5 *Magic squares*. An  $n \times n$  matrix that is filled with the numbers 1, 2, 3, ...,  $n^2$  is a magic square if the sum of the elements in each row, in each column, and in the two diagonals is the same value.

Write a program that reads in 16 values from the keyboard and tests whether they form a magic square when put into a  $4 \times 4$  array. You need to test two features:

|    |    |    |    |
|----|----|----|----|
| 16 | 3  | 2  | 13 |
| 5  | 10 | 11 | 8  |
| 9  | 6  | 7  | 12 |
| 4  | 15 | 14 | 1  |

1. Does each of the numbers 1, 2, ..., 16 occur in the user input?

2. When the numbers are put into a square, are the sums of the rows, columns, and diagonals equal to each other?

- P6.6 Implement the following algorithm to construct magic  $n \times n$  squares; it works only if  $n$  is odd.

```

Set row = n - 1, column = n / 2.
For k = 1 ... n * n
 Place k at [row][column].
 Increment row and column.
 If the row or column is n, replace it with 0.
 If the element at [row][column] has already been filled
 Set row and column to their previous values.
 Decrement row.

```

Here is the  $5 \times 5$  square that you get if you follow this method:

|    |    |    |    |    |
|----|----|----|----|----|
| 11 | 18 | 25 | 2  | 9  |
| 10 | 12 | 19 | 21 | 3  |
| 4  | 6  | 13 | 20 | 22 |
| 23 | 5  | 7  | 14 | 16 |
| 17 | 24 | 1  | 8  | 15 |

Write a program whose input is the number  $n$  and whose output is the magic square of order  $n$  if  $n$  is odd.

- P6.7 A theater seating chart is implemented as a two-dimensional array of ticket prices, like this:

```

10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
20 20 30 30 40 40 30 30 20 20
20 30 30 40 50 50 40 30 30 20
30 40 50 50 50 50 50 40 30

```

Write a program that prompts users to pick either a seat or a price. Mark sold seats by changing the price to 0. When a user specifies a seat, make sure it is available. When a user specifies a price, find any seat with that price.



© lepas2004/iStockphoto.

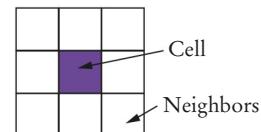
- P6.8 Write a program that plays tic-tac-toe. The tic-tac-toe game is played on a  $3 \times 3$  grid as in the photo at right. The game is played by two players, who take turns. The first player marks moves with a circle, the second with a cross. The player who has formed a horizontal, vertical, or diagonal sequence of three marks wins. Your program should draw the game board, ask the user for the coordinates of the next mark, change the players after every successful move, and pronounce the winner.



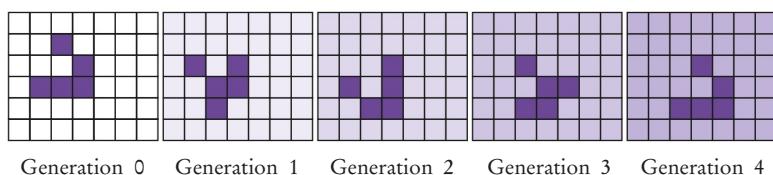
© KathyMuller/iStockphoto.

**P6.9** *The Game of Life* is a well-known mathematical game that gives rise to amazingly complex behavior, although it can be specified by a few simple rules. (It is not actually a game in the traditional sense, with players competing for a win.) Here are the rules. The game is played on a rectangular board. Each square can be either empty or occupied. At the beginning, you can specify empty and occupied cells in some way; then the game runs automatically. In each *generation*, the next generation is computed. A new cell is born on an empty square if it is surrounded by exactly three occupied neighbor cells. A cell dies of overcrowding if it is surrounded by four or more neighbors, and it dies of loneliness if it is surrounded by zero or one neighbor. A neighbor is an occupant of an adjacent square to the left, right, top, or bottom or in a diagonal direction. Figure 20 shows a cell and its neighbor cells.

Many configurations show interesting behavior when subjected to these rules. Figure 21 shows a *glider*, observed over five generations. After four generations, it is transformed into the identical shape, but located one square to the right and below.



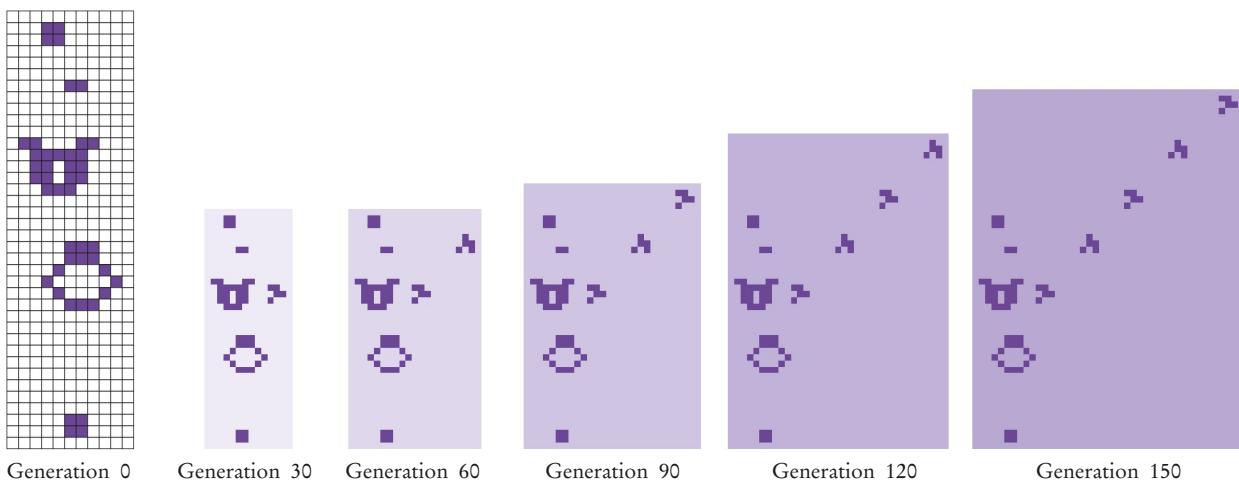
**Figure 20**  
Neighborhood of a Cell



**Figure 21** Glider

One of the more amazing configurations is the glider gun: a complex collection of cells that, after 30 moves, turns back into itself and a glider (see Figure 22).

Program the game to eliminate the drudgery of computing successive generations by hand. Use a two-dimensional array to store the rectangular configuration. Write a program that shows successive generations of the game. Ask the user to specify the original configuration, by typing in a configuration of spaces and o characters.



**Figure 22** Glider Gun

- Business P6.10** A pet shop wants to give a discount to its clients if they buy one or more pets and at least five other items. The discount is equal to 20 percent of the cost of the other items, but not the pets.

Implement a method

```
public static void discount(double[] prices, boolean[] isPet, int nItems)
```

The method receives information about a particular sale. For the  $i$ th item, `prices[i]` is the price before any discount, and `isPet[i]` is true if the item is a pet.

Write a program that prompts a cashier to enter each price and then a `Y` for a pet or `N` for another item. Use a price of `-1` as a sentinel. Save the inputs in an array. Call the method that you implemented, and display the discount.



© joshblake/iStockphoto.

- Business P6.11** A supermarket wants to reward its best customer of each day, showing the customer's name on a screen in the supermarket. For that purpose, the customer's purchase amount is stored in an `ArrayList<Double>` and the customer's name is stored in a corresponding `ArrayList<String>`.

Implement a method

```
public static String nameOfBestCustomer(ArrayList<Double> sales,
 ArrayList<String> customers)
```

that returns the name of the customer with the largest sale.

Write a program that prompts the cashier to enter all prices and names, adds them to two array lists, calls the method that you implemented, and displays the result. Use a price of `0` as a sentinel.

- Business P6.12** Improve the program of Exercise P6.11 so that it displays the top customers, that is, the `topN` customers with the largest sales, where `topN` is a value that the user of the program supplies.

Implement a method

```
public static ArrayList<String> nameOfBestCustomers(ArrayList<Double> sales,
 ArrayList<String> customers, int topN)
```

If there were fewer than `topN` customers, include all of them.

- Science P6.13** Sounds can be represented by an array of “sample values” that describe the intensity of the sound at a point in time. The program `ch06/sound/SoundEffect.java` reads a sound file (in WAV format), calls a method `process` for processing the sample values, and saves the sound file. Your task is to implement the `process` method by introducing an echo. For each sound value, add the value from 0.2 seconds ago. Scale the result so that no value is larger than 32767.



© GordonHeeley/iStockphoto.

- Science P6.14** You are given a two-dimensional array of values that give the height of a terrain at different points in a square. Write a method

```
public static void floodMap(double[][] heights, double waterLevel)
```

that prints out a flood map, showing which of the points in the terrain would be flooded if the water level was the given value. In the flood map, print a \* for each flooded point and a space for each point that is not flooded.

Here is a sample map:

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



© nicolamargate/iStockphoto

Then write a program that reads one hundred terrain height values and shows how the terrain gets flooded when the water level increases in ten steps from the lowest point in the terrain to the highest.

- Science P6.15** Sample values from an experiment often need to be smoothed out. One simple approach is to replace each value in an array with the average of the value and its two neighboring values (or one neighboring value if it is at either end of the array). Implement a method

```
public static void smooth(double[] values, int size)
```

that carries out this operation. You should not create another array in your solution.

- Science P6.16** Modify the ch06/animation/BlockAnimation.java program to show an animated sine wave. In the  $i$ th frame, shift the sine wave by  $i$  degrees.

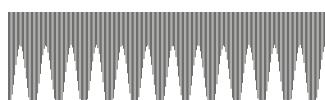
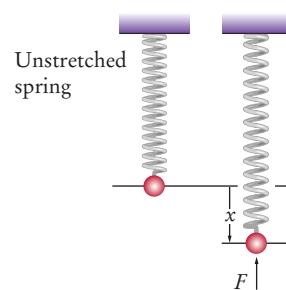
- Science P6.17** Write a program that models the movement of an object with mass  $m$  that is attached to an oscillating spring. When a spring is displaced from its equilibrium position by an amount  $x$ , Hooke's law states that the restoring force is

$$F = -kx$$

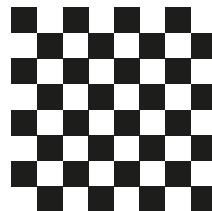
where  $k$  is a constant that depends on the spring. (Use 10 N/m for this simulation.)

Start with a given displacement  $x$  (say, 0.5 meter). Set the initial velocity  $v$  to 0. Compute the acceleration  $a$  from Newton's law ( $F = ma$ ) and Hooke's law, using a mass of 1 kg. Use a small time interval  $\Delta t = 0.01$  second. Update the velocity—it changes by  $a\Delta t$ . Update the displacement—it changes by  $v\Delta t$ .

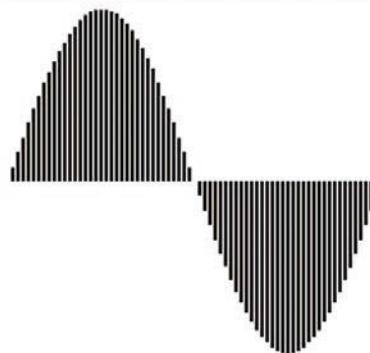
Every ten iterations, plot the spring displacement as a bar, where 1 pixel represents 1 cm. Use the technique in Special Topic 4.3 for creating an image.



- ■ **Graphics P6.18** Using the technique of Special Topic 4.3, generate the image of a checkerboard.



- **Graphics P6.19** Using the technique of Special Topic 4.3, generate the image of a sine wave. Draw a line of pixels for every five degrees.



### ANSWERS TO SELF-CHECK QUESTIONS

- 1.**

```
int[] primes = { 2, 3, 5, 7, 11 };
```
  - 2.** 2, 3, 5, 3, 2
  - 3.** 3, 4, 6, 8, 12
  - 4.**

```
values[0] = 10;
values[9] = 10;
```

  
or better: 

```
values[values.length - 1] = 10;
```
  - 5.**

```
String[] words = new String[10];
```
  - 6.**

```
String[] words = { "Yes", "No" };
```
  - 7.** No. Because you don't store the values, you need to print them when you read them. But you don't know where to add the `<=` until you have seen all values.
  - 8.** It counts how many elements of `values` are zero.
  - 9.**

```
for (double x : values)
{
 System.out.println(x);
}
```
  - 10.**

```
double product = 1;
for (double f : factors)
{
 product = product * f;
}
```
  - 11.** The loop writes a value into `values[i]`. The enhanced for loop does not have the index variable `i`.
  - 12.**

```
20 <= largest value
10
20 <= largest value
```
  - 13.**

```
int count = 0;
for (double x : values)
{
 if (x == 0) { count++; }
```
  - 14.** If all elements of `values` are negative, then the result is incorrectly computed as 0.
  - 15.**

```
for (int i = 0; i < values.length; i++)
{
 System.out.print(values[i]);
 if (i < values.length - 1)
 {
 System.out.print(" | ");
 }
}
```
- Now you know why we set up the loop the other way.
- 16.** If the array has no elements, then the program terminates with an exception.

- 17.** If there is a match, then pos is incremented before the loop exits.
- 18.** This loop sets all elements to values[pos].
- 19.** int[] numbers = squares(5);
- 20.** public static void fill(int[] values, int value)
 {
 for (int i = 0; i < values.length; i++)
 {
 values[i] = value;
 }
 }
- 21.** The method returns an array whose length is given in the first argument. The array is filled with random integers between 0 and n – 1.
- 22.** The contents of scores is unchanged. The reverse method returns a new array with the reversed numbers.

**23.**

| values    | result    | i |
|-----------|-----------|---|
| [1, 4, 9] | [0, 0, 0] | 0 |
|           | [9, 0, 0] | X |
|           | [9, 4, 0] | 2 |
|           | [9, 4, 1] |   |

- 24.** Use the first algorithm. The order of elements does not matter when computing the sum.

- 25. Find the minimum value.**

Calculate the sum.

Subtract the minimum value from the sum.

- 26.** Use the algorithm for counting matches (Section 4.7.2) twice, once for counting the positive values and once for counting the negative values.

- 27.** You need to modify the algorithm in Section 6.3.4.

```
boolean first = true;
for (int i = 0; i < values.length; i++)
{
 if (values[i] > 0)
 {
 if (first) { first = false; }
 else { System.out.print(", "); }
 }
 System.out.print(values[i]);
}
```

Note that you can no longer use  $i > 0$  as the criterion for printing a separator.

- 28.** Use the algorithm to collect all positive elements in an array, then use the algorithm in Section 6.3.4 to print the array of matches.
- 29.** The paperclip for  $i$  assumes positions 0, 1, 2, 3. When  $i$  is incremented to 4, the condition  $i < \text{size} / 2$  becomes false, and the loop ends. Similarly, the paperclip for  $j$  assumes positions 4, 5, 6, 7, which are the valid positions for the second half of the array.



- 30.** It reverses the elements in the array.
- 31.** Here is one solution. The basic idea is to move all odd elements to the end. Put one paper clip at the beginning of the array and one at the end. If the element at the first paper clip is odd, swap it with the one at the other paper clip and move that paper clip to the left. Otherwise, move the first paper clip to the right. Stop when the two paper clips meet. Here is the pseudocode:

```
i = 0
j = size - 1
While i < j
 If a[i] is odd
 Swap elements at positions i and j.
 j--
 Else
 i++
```

- 32.** Here is one solution. The idea is to remove all odd elements and move them to the end. The trick is to know when to stop. Nothing is gained by moving odd elements into the area that already contains moved elements, so we want to mark that area with another paper clip.

```
i = 0
moved = size
While i < moved
 If a[i] is odd
 Remove the element at position i.
 Add the removed element to the end.
 moved--
```

- 33.** When you read inputs, you get to see values one at a time, and you can't peek ahead. Picking cards one at a time from a deck of cards

- simulates this process better than looking at a sequence of items, all of which are revealed.
- 34.** You get the total number of gold, silver, and bronze medals in the competition. In our example, there are five of each.
- 35.**
- ```
for (int i = 0; i < 8; i++)
{
    for (int j = 0; j < 8; j++)
    {
        board[i][j] = (i + j) % 2;
    }
}
```
- 36.** `String[][] board = new String[3][3];`
- 37.** `board[0][2] = "x";`
- 38.** `board[0][0], board[1][1], board[2][2]`
- 39.**
- ```
ArrayList<Integer> primes =
 new ArrayList<Integer>();
primes.add(2);
primes.add(3);
primes.add(5);
primes.add(7);
primes.add(11);
```
- 40.**
- ```
for (int i = primes.size() - 1; i >= 0; i--)
{
    System.out.println(primes.get(i));
}
```
- 41.** "Ann", "Cal"
- 42.** The names variable has not been initialized.
- 43.** names1 contains "Emily", "Bob", "Cindy", "Dave"; names2 contains "Dave".
- 44.** Because the number of weekdays doesn't change, there is no disadvantage to using an array, and it is easier to initialize:
- ```
String[] weekdayNames = { "Monday", "Tuesday",
 "Wednesday", "Thursday", "Friday",
 "Saturday", "Sunday" };
```
- 45.** Reading inputs into an array list is much easier.

## WORKED EXAMPLE 6.1

**Rolling the Dice**

**Problem Statement** Your task is to analyze whether a die is fair by counting how often the values 1, 2, ..., 6 appear. Your input is a sequence of die toss values, and you should print a table with the frequencies of each die value.



© ktsimage/iStockphoto.

**Step 1** Decompose your task into steps.

Our first try at decomposition simply echoes the problem statement:

- Read the die values into an array.**
- Count how often the values 1, 2, ..., 6 appear in the array.**
- Print the counts.**

But let's think about the task a little more. This decomposition suggests that we first read and store all die values. Do we really need to store them? After all, we only want to know how often each face value appears. If we keep an array of counters, we can discard each input after incrementing the counter.

This refinement yields the following outline:

- For each input value i**
- Increment the counter corresponding to i.**
- Print the counters.**

**Step 2** Determine which algorithm(s) you need.

We don't have a ready-made algorithm for reading inputs and incrementing a counter, but it is straightforward to develop one. Suppose we read an input into `value`. This is an integer between 1 and 6. If we have an array `counters` of length 6, then we simply call

```
counters[value - 1]++;
```

Alternatively, we can use an array of seven integers, “wasting” the element `counters[0]`. That trick makes it easier to update the counters. When reading an input value, we simply execute

```
counters[value]++; // value is between 1 and 6
```

That is, we declare the array as

```
int[] counters = new int[sides + 1];
```

Why introduce a `sides` variable? Suppose you later changed your mind and wanted to investigate 12-sided dice:



© Ryan Ruffatti/iStockphoto.

Then the program can simply be changed by setting `sides` to 12.

## WE2 Chapter 6 Arrays and Array Lists

The only remaining task is to print the counts. A typical output might look like this:

```
1: 3
2: 3
3: 2
4: 2
5: 2
6: 0
```

We haven't seen an algorithm for this exact output format. It is similar to the basic loop for printing all elements:

```
for (int element : counters)
{
 System.out.println(element);
}
```

However, that loop is not appropriate for two reasons. First, it displays the unused 0 entry. The "enhanced" for loop is no longer suitable if we want to skip that entry. We need a traditional for loop instead:

```
for (int i = 1; i < counters.length; i++)
{
 System.out.println(counters[i]);
}
```

This loop prints the counter values, but it doesn't quite match the sample output. We also want the corresponding face values:

```
for (int i = 1; i < counters.length; i++)
{
 System.out.printf("%2d: %4d%n", i, counters[i]);
}
```

### Step 3 Use methods to structure your program.

We will provide a method for each step:

- `int[] countInputs(int sides)`
- `printCounters(int[] counters)`

The `main` method calls these methods:

```
int[] counters = countInputs(6);
printCounters(counters);
```

The `countInputs` method reads all inputs, increments the matching counters, and returns the array of counters. The `printCounters` method prints the value of the faces and counters, as already described.

### Step 4 Assemble and test the program.

The listing at the end of this section shows the complete program. There is one notable feature that we have not previously discussed. When updating a counter

```
counters[value]++;
```

we want to be sure that the user did not provide a wrong input which would cause an array bounds error. Therefore, we reject inputs < 1 or > `sides`.

The following table shows test cases and their expected output. To save space, we only show the counters in the output.

| Test Case       | Expected Output | Comment                                                  |
|-----------------|-----------------|----------------------------------------------------------|
| 1 2 3 4 5 6     | 1 1 1 1 1 1     | Each number occurs once.                                 |
| 1 2 3           | 1 1 1 0 0 0     | Numbers that don't appear should have counts of zero.    |
| 1 2 3 1 2 3 4   | 2 2 2 1 0 0     | The counters should reflect how often each input occurs. |
| (No input)      | 0 0 0 0 0 0     | This is a legal input; all counters are zero.            |
| 0 1 2 3 4 5 6 7 | <b>Error</b>    | Each input should be between 1 and 6.                    |

Here's the complete program:

### worked\_example\_1/Dice.java

```

1 import java.util.Scanner;
2 /**
3 * This program reads a sequence of die toss values and prints how many times
4 * each value occurred.
5 */
6 public class Dice
7 {
8 public static void main(String[] args)
9 {
10 int[] counters = countInputs(6);
11 printCounters(counters);
12 }
13
14 /**
15 * Reads a sequence of die toss values between 1 and sides (inclusive)
16 * and counts how frequently each of them occurs.
17 * @return an array whose ith element contains the number of
18 * times the value i occurred in the input. The 0 element is unused.
19 */
20 public static int[] countInputs(int sides)
21 {
22 int[] counters = new int[sides + 1]; // counters[0] is not used
23
24 System.out.println("Please enter values, Q to quit:");
25 Scanner in = new Scanner(System.in);
26 while (in.hasNextInt())
27 {
28 int value = in.nextInt();
29
30 // Increment the counter for the input value
31
32 if (1 <= value && value <= sides)
33 {
34 counters[value]++;
35 }
36 }
37
38 for (int i = 1; i <= sides; i++)
39 {
40 System.out.print(i + " " + counters[i] + " ");
41 }
42 System.out.println();
43 }
44
45 }
```

## WE4 Chapter 6 Arrays and Array Lists

```
36 }
37 else
38 {
39 System.out.println(value + " is not a valid input.");
40 }
41 }
42 return counters;
43 }
44
45 /**
46 Prints a table of die value counters.
47 @param counters an array of counters.
48 counters[0] is not printed.
49 */
50 public static void printCounters(int[] counters)
51 {
52 for (int j = 1; j < counters.length; j++)
53 {
54 System.out.println(j + ": " + counters[j]);
55 }
56 }
57 }
```

### Program Run

```
Please enter values, Q to quit:
1 2 3 1 2 3 4 Q
1: 2
2: 2
3: 2
4: 1
5: 0
6: 0
```

## WORKED EXAMPLE 6.2

## A World Population Table



**Problem Statement** You are to print the following population data in tabular format and add column totals that show the total world populations in the given years.

| Year          | Population Per Continent (in millions) |      |      |      |      |      |      |  |
|---------------|----------------------------------------|------|------|------|------|------|------|--|
|               | 1750                                   | 1800 | 1850 | 1900 | 1950 | 2000 | 2050 |  |
| Africa        | 106                                    | 107  | 111  | 133  | 221  | 767  | 1766 |  |
| Asia          | 502                                    | 635  | 809  | 947  | 1402 | 3634 | 5268 |  |
| Australia     | 2                                      | 2    | 2    | 6    | 13   | 30   | 46   |  |
| Europe        | 163                                    | 203  | 276  | 408  | 547  | 729  | 628  |  |
| North America | 2                                      | 7    | 26   | 82   | 172  | 307  | 392  |  |
| South America | 16                                     | 24   | 38   | 74   | 167  | 511  | 809  |  |

**Step 1** First, we break down the task into steps:

- Initialize the table data.
- Print the table.
- Compute and print the column totals.

**Step 2** Initialize the table as a sequence of rows:

```
int[][] populations =
{
 { 106, 107, 111, 133, 221, 767, 1766 },
 { 502, 635, 809, 947, 1402, 3634, 5268 },
 { 2, 2, 2, 6, 13, 30, 46 },
 { 163, 203, 276, 408, 547, 729, 628 },
 { 2, 7, 26, 82, 172, 307, 392 },
 { 16, 24, 38, 74, 167, 511, 809 }
};
```

**Step 3** To print the row headers, we also need a one-dimensional array of the continent names. Note that it has the same number of rows as our table.

```
String[] continents =
{
 "Africa",
 "Asia",
 "Australia",
 "Europe",
 "North America",
 "South America"
};
```

## WE6 Chapter 6 Arrays and Array Lists

To print a row, we first print the continent name, then all columns. This is achieved with two nested loops. The outer loop prints each row:

```
// Print population data
for (int i = 0; i < ROWS; i++)
{
 // Print the ith row
 .
 System.out.println(); // Start a new line at the end of the row
}
```

To print a row, we first print the row header, then all columns:

```
System.out.printf("%20s", continents[i]);
for (int j = 0; j < COLUMNS; j++)
{
 System.out.printf("%5d", populations[i][j]);
}
```

**Step 4** To print the column sums, we use the algorithm that was described in Section 6.7.4. We carry out that computation once for each column.

```
for (int j = 0; j < COLUMNS; j++)
{
 int total = 0;
 for (int i = 0; i < ROWS; i++)
 {
 total = total + populations[i][j];
 }
 System.out.printf("%5d", total);
}
```

Here is the complete program:

### [worked\\_example\\_2/WorldPopulation.java](#)

```
1 /**
2 * This program prints a table showing the world population growth over 300 years.
3 */
4 public class WorldPopulation
5 {
6 public static void main(String[] args)
7 {
8 final int ROWS = 6;
9 final int COLUMNS = 7;
10
11 int[][] populations =
12 {
13 { 106, 107, 111, 133, 221, 767, 1766 },
14 { 502, 635, 809, 947, 1402, 3634, 5268 },
15 { 2, 2, 2, 6, 13, 30, 46 },
16 { 163, 203, 276, 408, 547, 729, 628 },
17 { 2, 7, 26, 82, 172, 307, 392 },
18 { 16, 24, 38, 74, 167, 511, 809 }
19 };
20
21 String[] continents =
22 {
23 "Africa",
24 "Asia",
25 "Australia",
```

```

26 "Europe",
27 "North America",
28 "South America"
29 };
30
31 System.out.println("Year 1750 1800 1850 1900 1950 2000 2050");
32
33 // Print population data
34
35 for (int i = 0; i < ROWS; i++)
36 {
37 // Print the ith row
38 System.out.printf("%20s", continents[i]);
39 for (int j = 0; j < COLUMNS; j++)
40 {
41 System.out.printf("%5d", populations[i][j]);
42 }
43 System.out.println(); // Start a new line at the end of the row
44 }
45
46 // Print column totals
47
48 System.out.print("World");
49 for (int j = 0; j < COLUMNS; j++)
50 {
51 int total = 0;
52 for (int i = 0; i < ROWS; i++)
53 {
54 total = total + populations[i][j];
55 }
56 System.out.printf("%5d", total);
57 }
58 System.out.println();
59 }
60 }
```

**Program Run**

|               | Year | 1750 | 1800 | 1850 | 1900 | 1950 | 2000 | 2050 |
|---------------|------|------|------|------|------|------|------|------|
| Africa        |      | 106  | 107  | 111  | 133  | 221  | 767  | 1766 |
| Asia          |      | 502  | 635  | 809  | 947  | 1402 | 3634 | 5268 |
| Australia     |      | 2    | 2    | 2    | 6    | 13   | 30   | 46   |
| Europe        |      | 163  | 203  | 276  | 408  | 547  | 729  | 628  |
| North America |      | 2    | 7    | 26   | 82   | 172  | 307  | 392  |
| South America |      | 16   | 24   | 38   | 74   | 167  | 511  | 809  |
| World         |      | 791  | 978  | 1262 | 1650 | 2522 | 5978 | 8909 |



# INPUT/OUTPUT AND EXCEPTION HANDLING

## CHAPTER GOALS

- To read and write text files
- To process command line arguments
- To throw and catch exceptions
- To implement programs that propagate checked exceptions

## CHAPTER CONTENTS

### 7.1 READING AND WRITING

#### TEXT FILES 332

- CE1** Backslashes in File Names 335
- CE2** Constructing a Scanner with a String 335
- ST1** Reading Web Pages 335
- ST2** File Dialog Boxes 335
- ST3** Reading and Writing Binary Data 336

### 7.2 TEXT INPUT AND OUTPUT 337

- ST4** Regular Expressions 344
- ST5** Reading an Entire File 344
- VE1** Computing a Document's Readability 

### 7.3 COMMAND LINE ARGUMENTS 345

- HT1** Processing Text Files 348
- WE1** Analyzing Baby Names 
- C&S** Encryption Algorithms 351

### 7.4 EXCEPTION HANDLING 352

- SYN** Throwing an Exception 352
- SYN** Catching Exceptions 354
- SYN** The throws Clause 357
- SYN** The try-with-resources Statement 357



James King-Holmes/Bletchley ParkTrust/Photo Researchers, Inc.

- PT1** Throw Early, Catch Late 359
- PT2** Do Not Squelch Exceptions 359
- PT3** Do Throw Specific Exceptions 360
- ST6** Assertions 360
- ST7** The try/finally Statement 360
- C&S** The Ariane Rocket Incident 361

### 7.5 APPLICATION: HANDLING

#### INPUT ERRORS 361

- VE2** Detecting Accounting Fraud 



James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

In this chapter, you will learn how to read and write files—a very useful skill for processing real world data. As an application, you will learn how to encrypt data. (The Enigma machine shown here is an encryption device used by Germany in World War II. Pioneering British computer scientists broke the code and were able to intercept encoded messages, which was a significant help in winning the war.) The remainder of this chapter tells you how your programs can report and recover from problems, such as missing files or malformed content, using the exception-handling mechanism of the Java language.

## 7.1 Reading and Writing Text Files

Use the Scanner class for reading text files.

We begin this chapter by discussing the common task of reading and writing files that contain text. Examples of text files include not only files that are created with a simple text editor, such as Windows Notepad, but also Java source code and HTML files.

In Java, the most convenient mechanism for reading text is to use the `Scanner` class. You already know how to use a `Scanner` for reading console input. To read input from a disk file, the `Scanner` class relies on another class, `File`, which describes disk files and directories. (The `File` class has many methods that we do not discuss in this book; for example, methods that delete or rename a file.)

To begin, construct a `File` object with the name of the input file:

```
File inputFile = new File("input.txt");
```

Then use the `File` object to construct a `Scanner` object:

```
Scanner in = new Scanner(inputFile);
```

This `Scanner` object reads text from the file `input.txt`. You can use the `Scanner` methods (such as `nextInt`, `nextDouble`, and `next`) to read data from the input file.

For example, you can use the following loop to process numbers in the input file:

```
while (in.hasNextDouble())
{
 double value = in.nextDouble();
 Process value.
}
```

When writing text files, use the `PrintWriter` class and the `print`/`println`/`printf` methods.

To write output to a file, you construct a `PrintWriter` object with the desired file name, for example

```
PrintWriter out = new PrintWriter("output.txt");
```

If the output file already exists, it is emptied before the new data are written into it. If the file doesn't exist, an empty file is created.

The `PrintWriter` class is an enhancement of the `PrintStream` class that you already know—`System.out` is a `PrintStream` object. You can use the familiar `print`, `println`, and `printf` methods with any `PrintWriter` object:

```
out.println("Hello, World!");
out.printf("Total: %.2f%n", total);
```

Close all files  
when you are done  
processing them.

When you are done processing a file, be sure to *close* the Scanner or PrintWriter:

```
in.close();
out.close();
```

If your program exits without closing the PrintWriter, some of the output may not be written to the disk file.

The following program puts these concepts to work. It reads a file containing numbers, and writes the numbers to another file, lined up in a column and followed by their total.

For example, if the input file has the contents

```
32 54 67.5 29 35 80
115 44.5 100 65
```

then the output file is

```
32.00
54.00
67.50
29.00
35.00
80.00
115.00
44.50
100.00
65.00
Total: 622.00
```

There is one additional issue that we need to tackle. If the input or output file for a Scanner doesn't exist, a `FileNotFoundException` occurs when the Scanner object is constructed. The compiler insists that we specify what the program should do when that happens. Similarly, the PrintWriter constructor generates this exception if it cannot open the file for writing. (This can happen if the name is illegal or the user does not have the authority to create a file in the given location.) In our sample program, we want to terminate the `main` method if the exception occurs. To achieve this, we label the `main` method with a `throws` declaration:

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

You will see in Section 7.4 how to deal with exceptions in a more professional way.

The `File`, `PrintWriter`, and `FileNotFoundException` classes are contained in the `java.io` package.

### sec01/Total.java

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 /**
7 * This program reads a file with numbers, and writes the numbers to another
8 * file, lined up in a column and followed by their total.
9 */
10 public class Total
11 {
12 public static void main(String[] args) throws FileNotFoundException
13 {
```

```

14 // Prompt for the input and output file names
15
16 Scanner console = new Scanner(System.in);
17 System.out.print("Input file: ");
18 String inputFileName = console.nextLine();
19 System.out.print("Output file: ");
20 String outputFileName = console.nextLine();
21
22 // Construct the Scanner and PrintWriter objects for reading and writing
23
24 File inputFile = new File(inputFileName);
25 Scanner in = new Scanner(inputFile);
26 PrintWriter out = new PrintWriter(outputFileName);
27
28 // Read the input and write the output
29
30 double total = 0;
31
32 while (in.hasNextDouble())
33 {
34 double value = in.nextDouble();
35 out.printf("%15.2f\n", value);
36 total = total + value;
37 }
38
39 out.printf("Total: %.2f\n", total);
40
41 in.close();
42 out.close();
43 }
44 }
```



- What happens when you supply the same name for the input and output files to the `Total` program? Try it out if you are not sure.
- What happens when you supply the name of a nonexistent input file to the `Total` program? Try it out if you are not sure.
- Suppose you wanted to add the total to an existing file instead of writing a new file. Self Check 1 indicates that you cannot simply do this by specifying the same file for input and output. How can you achieve this task? Provide the pseudo-code for the solution.
- How do you modify the program so that it shows the average, not the total, of the inputs?
- How can you modify the `Total` program so that it writes the values in two columns, like this:

|                      |       |
|----------------------|-------|
| 32.00                | 54.00 |
| 67.50                | 29.00 |
| 35.00                | 80.00 |
| 115.00               | 44.50 |
| 100.00               | 65.00 |
| <b>Total:</b> 622.00 |       |

**Practice It** Now you can try these exercises at the end of the chapter: R7.1, R7.2, E7.1.

**Common Error 7.1****Backslashes in File Names**

When you specify a file name as a string literal, and the name contains backslash characters (as in a Windows file name), you must supply each backslash twice:

```
File inputFile = new File("c:\\homework\\input.dat");
```

A single backslash inside a quoted string is an **escape character** that is combined with the following character to form a special meaning, such as `\n` for a newline character. The `\\" combination denotes a single backslash.`

When a user supplies a file name to a program, however, the user should not type the backslash twice.

**Common Error 7.2****Constructing a Scanner with a String**

When you construct a `PrintWriter` with a string, it writes to a file:

```
PrintWriter out = new PrintWriter("output.txt");
```

However, this does *not* work for a `Scanner`. The statement

```
Scanner in = new Scanner("input.txt"); // Error?
```

does *not* open a file. Instead, it simply reads through the string: `in.next()` returns the string "input.txt". (This is occasionally useful—see Section 7.2.4.)

You must simply remember to use `File` objects in the `Scanner` constructor:

```
Scanner in = new Scanner(new File("input.txt")); // OK
```

**Special Topic 7.1****Reading Web Pages**

You can read the contents of a web page with this sequence of commands:

```
String address = "http://horstmann.com/index.html";
URL pageLocation = new URL(address);
Scanner in = new Scanner(pageLocation.openStream());
```

Now simply read the contents of the web page with the `Scanner` in the usual way. The `URL` constructor and the `openStream` method can throw an `IOException`, so you need to tag the `main` method with `throws IOException`. (See Section 7.4.3 for more information on the `throws` clause.)

The `URL` class is contained in the `java.net` package.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program that reads data from a web page.

**Special Topic 7.2****File Dialog Boxes**

In a program with a graphical user interface, you will want to use a file dialog box (such as the one shown in the figure below) whenever the users of your program need to pick a file. The `JFileChooser` class implements a file dialog box for the Swing user-interface toolkit.

The `JFileChooser` class has many options to fine-tune the display of the dialog box, but in its most basic form it is quite simple: Construct a file chooser object; then call the `showOpenDialog` or `showSaveDialog` method. Both methods show the same dialog box, but the button for selecting a file is labeled "Open" or "Save", depending on which method you call.

For better placement of the dialog box on the screen, you can specify the user-interface component over which to pop up the dialog box. If you don't care where the dialog box pops

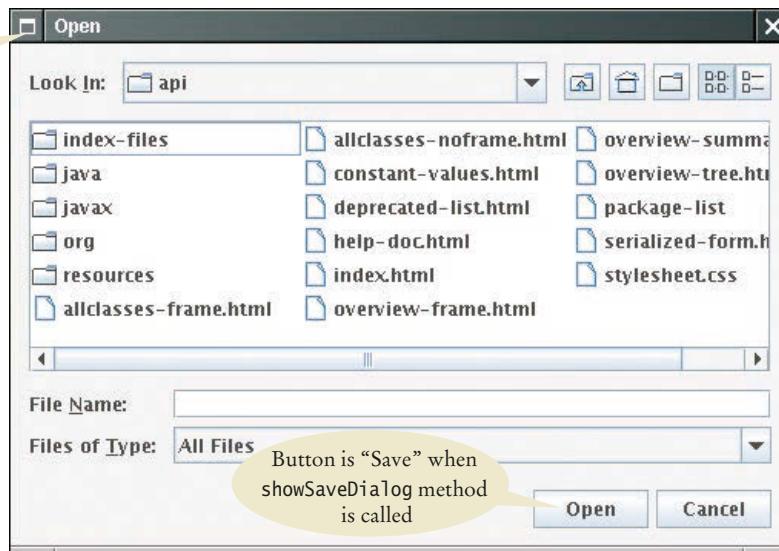
up, you can simply pass `null`. The `showOpenDialog` and `showSaveDialog` methods return either `JFileChooser.APPROVE_OPTION`, if the user has chosen a file, or `JFileChooser.CANCEL_OPTION`, if the user canceled the selection. If a file was chosen, then you call the `getSelectedFile` method to obtain a `File` object that describes the file. Here is a complete example:

```
JFileChooser chooser = new JFileChooser();
Scanner in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
{
 File selectedFile = chooser.getSelectedFile();
 in = new Scanner(selectedFile);
 ...
}
```

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program that demonstrates how to use a file chooser.

Call with  
showOpenDialog  
method



A `JFileChooser` Dialog Box

### Special Topic 7.3



### Reading and Writing Binary Data

You use the `Scanner` and `PrintWriter` classes to read and write text files. Text files contain sequences of characters. Other files, such as images, are not made up of characters but of bytes. A `byte` is a fundamental storage unit in a computer—a number consisting of eight binary digits. (A byte can represent unsigned integers between 0 and 255 or signed integers between -128 and 127.) The Java library has a different set of classes, called streams, for working with binary files. While modifying binary files is quite challenging and beyond the scope of this book, we give you a simple example of copying binary data from a web site to a file.

You use an `InputStream` to read binary data. For example,

```
URL imageLocation = new URL("http://horstmann.com/java4everyone/duke.gif");
InputStream in = imageLocation.openStream();
```

To write binary data to a file, use a `FileOutputStream`:

```
FileOutputStream out = new FileOutputStream("duke.gif");
```

The `read` method of an input stream reads a single byte and returns `-1` when no further input is available. The `write` method of an output stream writes a single byte.

The following loop copies all bytes from an input stream to an output stream:

```
boolean done = false;
while (!done)
{
 int input = in.read(); // -1 or a byte between 0 and 255
 if (input == -1) { done = true; }
 else { out.write(input); }
}
```

## 7.2 Text Input and Output

In the following sections, you will learn how to process text with complex contents, and you will learn how to cope with challenges that often occur with real data.

### 7.2.1 Reading Words

The next method reads a string that is delimited by white space.

The next method of the Scanner class reads the next string. Consider the loop

```
while (in.hasNext())
{
 String input = in.next();
 System.out.println(input);
}
```

If the user provides the input:

Mary had a little lamb

this loop prints each word on a separate line:

```
Mary
had
a
little
lamb
```

However, the words can contain punctuation marks and other symbols. The next method returns any sequence of characters that is not white space. *White space* includes spaces, tab characters, and the newline characters that separate lines. For example, the following strings are considered “words” by the next method:

```
snow.
1729
C++
```

(Note the period after snow—it is considered a part of the word because it is not white space.)

Here is precisely what happens when the next method is executed. Input characters that are white space are *consumed*—that is, removed from the input. However, they do not become part of the word. The first character that is not white space becomes the first character of the word. More characters are added until either another white space character occurs, or the end of the input file has been reached. However, if the end of the input file is reached before any character was added to the word, a “no such element exception” occurs.

Sometimes, you want to read just the words and discard anything that isn't a letter. You achieve this task by calling the `useDelimiter` method on your `Scanner` object:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

Here, we set the character pattern that separates words to “any sequence of characters other than letters”. (See Special Topic 7.4.) With this setting, punctuation and numbers are not included in the words returned by the `next` method.

## 7.2.2 Reading Characters

Sometimes, you want to read a file one character at a time. You will see an example in Section 7.3 where we encrypt the characters of a file. You achieve this task by calling the `useDelimiter` method on your `Scanner` object with an empty string:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
```

Now each call to `next` returns a string consisting of a single character. Here is how you can process the characters:

```
while (in.hasNext())
{
 char ch = in.next().charAt(0);
 Process ch.
}
```

## 7.2.3 Classifying Characters

The `Character` class has methods for classifying characters.

When you read a character, or when you analyze the characters in a word or line, you often want to know what kind of character it is. The `Character` class declares several useful methods for this purpose. Each of them has an argument of type `char` and returns a `boolean` value (see Table 1).

For example, the call

```
Character.isDigit(ch)
```

returns `true` if `ch` is a digit ('0' . . . '9' or a digit in another writing system—see Computing & Society 2.2), `false` otherwise.

**Table 1** Character Testing Methods

| Method                    | Examples of Accepted Characters |
|---------------------------|---------------------------------|
| <code>isDigit</code>      | 0, 1, 2                         |
| <code>isLetter</code>     | A, B, C, a, b, c                |
| <code>isUpperCase</code>  | A, B, C                         |
| <code>isLowerCase</code>  | a, b, c                         |
| <code>isWhiteSpace</code> | space, newline, tab             |

## 7.2.4 Reading Lines

The `nextLine` method reads an entire line.

When each line of a file is a data record, it is often best to read entire lines with the `nextLine` method:

```
String line = in.nextLine();
```

The next input line (without the newline character) is placed into the string `line`. You can then take the line apart for further processing.

The `hasNextLine` method returns `true` if there is at least one more line in the input, `false` when all lines have been read. To ensure that there is another line to process, call the `hasNextLine` method before calling `nextLine`.

Here is a typical example of processing lines in a file. A file with population data from the CIA Fact Book site (<https://www.cia.gov/library/publications/the-world-factbook/index.html>) contains lines such as the following:

```
China 1330044605
India 1147995898
United States 303824646
...
```

Because some country names have more than one word, it would be tedious to read this file using the `next` method. For example, after reading `United`, how would your program know that it needs to read another word before reading the population count?

Instead, read each input line into a string:

```
while (in.hasNextLine())
{
 String line = nextLine();
 Process line.
}
```

Use the `isDigit` and `isWhiteSpace` methods introduced to find out where the name ends and the number starts.

Locate the first digit:

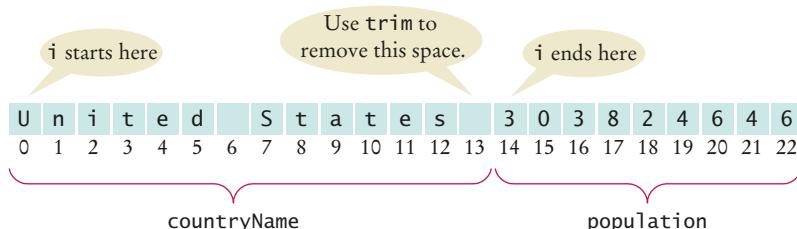
```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

Then extract the country name and population:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

However, the country name contains one or more spaces at the end. Use the `trim` method to remove them:

```
countryName = countryName.trim();
```



The `trim` method returns the string with all white space at the beginning and end removed.

There is one additional problem. The population is stored in a string, not a number. In Section 7.2.6, you will see how to convert the string to a number.

### 7.2.5 Scanning a String

In the preceding section, you saw how to break a string into parts by looking at individual characters. Another approach is occasionally easier. You can use a `Scanner` object to read the characters from a string:

```
Scanner lineScanner = new Scanner(line);
```

Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

```
String countryName = lineScanner.next(); // Read first word
// Add more words to countryName until number encountered
while (!lineScanner.hasNextInt())
{
 countryName = countryName + " " + lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

### 7.2.6 Converting Strings to Numbers

Sometimes you have a string that contains a number, such as the `population` string in Section 7.2.4. For example, suppose that the string is the character sequence "303824646". To get the integer value 303824646, you use the `Integer.parseInt` method:

```
int populationValue = Integer.parseInt(population);
// populationValue is the integer 303824646
```

To convert a string containing floating-point digits to its floating-point value, use the `Double.parseDouble` method. For example, suppose `input` is the string "3.95".

```
double price = Double.parseDouble(input);
// price is the floating-point number 3.95
```

You need to be careful when calling the `Integer.parseInt` and `Double.parseDouble` methods. The argument must be a string containing the digits of an integer, without any additional characters. Not even spaces are allowed! In our situation, we happen to know that there won't be any spaces at the beginning of the string, but there might be some at the end. Therefore, we use the `trim` method:

```
int populationValue = Integer.parseInt(population.trim());
```

How To 7.1 on page 348 continues this example.

If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

### 7.2.7 Avoiding Errors When Reading Numbers

You have used the `nextInt` and `nextDouble` methods of the `Scanner` class many times, but here we will have a look at what happens in “abnormal” situations. Suppose you call

```
int value = in.nextInt();
```

The `nextInt` method recognizes numbers such as 3 or -21. However, if the input is not a properly formatted number, an “input mismatch exception” occurs. For example, consider an input containing the characters

2 1 s t c e n t u r y

White space is consumed and the word 21st is read. However, this word is not a properly formatted number, causing an input mismatch exception in the nextInt method.

If there is no input at all when you call `nextInt` or `nextDouble`, a “no such element exception” occurs. To avoid exceptions, use the `hasNextInt` method to screen the input when reading an integer. For example,

```
if (in.hasNextInt())
{
 int value = in.nextInt();
 ...
}
```

Similarly, you should call the `hasNextDouble` method before calling `nextDouble`.

## 7.2.8 Mixing Number, Word, and Line Input

The `nextInt`, `nextDouble`, and `next` methods *do not* consume the white space that follows the number or word. This can be a problem if you alternate between calling `nextInt`/`nextDouble`/`next` and `nextLine`. Suppose a file contains country names and population values in this format:

China  
1330044605  
India  
1147995898  
United States  
303824646

Now suppose you read the file with these instructions:

```
while (in.hasNextLine())
{
 String countryName = in.nextLine();
 int population = in.nextInt();
 Process the country name and population.
}
```

Initially, the input contains

C h i n a \n 1 3 3 0 0 0 4 4 6 0 5 \n I n d i a \n

After the first call to the `nextLine` method, the input contains

```
1 3 3 0 0 4 4 6 0 5 \n I n d i a \n
```

After the call to `nextInt`, the input contains

\n T n d i a \n

Note that the `nextInt` call did *not* consume the newline character. Therefore, the second call to `nextLine` reads an empty string!

The remedy is to add a call to nextLine after reading the population value:

```
String countryName = in.nextLine();
int population = in.nextInt();
in.nextLine(); // Consume the newline
```

The call to `nextLine` consumes any remaining white space *and* the newline character.

### 7.2.9 Formatting Output

When you write numbers or strings, you often want to control how they appear. For example, dollar amounts are usually formatted with two significant digits, such as

Cookies: 3.20

You know from Section 2.3.2 how to achieve this output with the `printf` method. In this section, we discuss additional options of the `printf` method.

Suppose you need to print a table of items and prices, each stored in an array, such as this one:

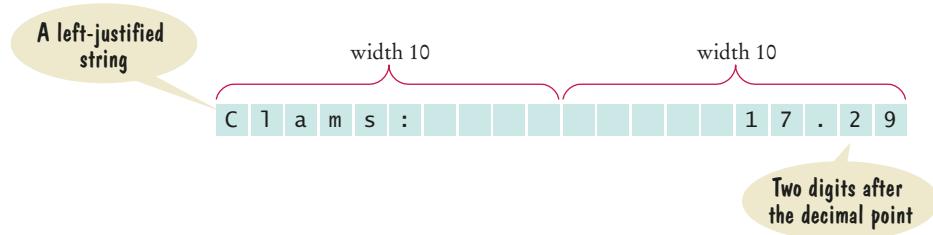
|           |       |
|-----------|-------|
| Cookies:  | 3.20  |
| Linguine: | 2.95  |
| Clams:    | 17.29 |

Note that the item strings line up to the left, whereas the numbers line up to the right. By default, the `printf` method lines up values to the right. To specify left alignment, you add a hyphen (-) before the field width:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

Here, we have two format specifiers.

- `%-10s` formats a left-justified string. The string `items[i] + ":"` is padded with spaces so it becomes ten characters wide. The - indicates that the string is placed on the left, followed by sufficient spaces to reach a width of 10.
- `%10.2f` formats a floating-point number, also in a field that is ten characters wide. However, the spaces appear to the left and the value to the right.



A construct such as `%-10s` or `%10.2f` is called a *format specifier*: it describes how a value should be formatted.

**Table 2** Format Flags

| Flag | Meaning                                 | Example                 |
|------|-----------------------------------------|-------------------------|
| -    | Left alignment                          | 1.23 followed by spaces |
| 0    | Show leading zeroes                     | 001.23                  |
| +    | Show a plus sign for positive numbers   | +1.23                   |
| (    | Enclose negative numbers in parentheses | (1.23)                  |
| ,    | Show decimal separators                 | 12,300                  |
| ^    | Convert letters to uppercase            | 1.23E+1                 |

**Table 3** Format Types

| Code | Type                                                                                         | Example |
|------|----------------------------------------------------------------------------------------------|---------|
| d    | Decimal integer                                                                              | 123     |
| f    | Fixed floating-point                                                                         | 12.30   |
| e    | Exponential floating-point                                                                   | 1.23e+1 |
| g    | General floating-point<br>(exponential notation is used for very large or very small values) | 12.3    |
| s    | String                                                                                       | Tax:    |

A format specifier has the following structure:

- The first character is a %
- Next, there are optional “flags” that modify the format, such as - to indicate left alignment. See Table 2 for the most common format flags.
- Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers.
- The format specifier ends with the format type, such as f for floating-point values or s for strings. There are quite a few format types—Table 3 shows the most important ones.


**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program that processes a file containing a mixture of text and numbers.

**SELF CHECK**


6. Suppose the input contains the characters Hello, World!. What are the values of word and input after this code fragment?  

```
String word = in.next();
String input = in.nextLine();
```
7. Suppose the input contains the characters 995.0 Fred. What are the values of number and input after this code fragment?  

```
int number = 0;
if (in.hasNextInt()) { number = in.nextInt(); }
String input = in.nextLine();
```
8. Suppose the input contains the characters 6E6 \$6,995.00. What are the values of x1 and x2 after this code fragment?  

```
double x1 = in.nextDouble();
double x2 = in.nextDouble();
```
9. Your input file contains a sequence of numbers, but sometimes a value is not available and is marked as N/A. How can you read the numbers and skip over the markers?
10. How can you remove spaces from the country name in Section 7.2.4 without using the trim method?

**Practice It** Now you can try these exercises at the end of the chapter: E7.4, E7.6, E7.7.

## Special Topic 7.4

**Regular Expressions**

A **regular expression** describes a character pattern. For example, numbers have a simple form. They contain one or more digits. The regular expression describing numbers is `[0-9]+`. The set `[0-9]` denotes any digit between 0 and 9, and the `+` means “one or more”.

The search commands of professional programming editors understand regular expressions. Moreover, several utility programs use regular expressions to locate matching text. A commonly used program that uses regular expressions is `grep` (which stands for “global regular expression print”). You can run grep from a command line or from inside some compilation environments. Grep is part of the UNIX operating system, and versions are available for Windows. It needs a regular expression and one or more files to search. When grep runs, it displays a set of lines that match the regular expression.

Suppose you want to find all magic numbers (see Programming Tip 2.2) in a file.

```
grep [0-9]+ Homework.java
```

lists all lines in the file `Homework.java` that contain sequences of digits. That isn’t terribly useful; lines with variable names `x1` will be listed. OK, you want sequences of digits that do *not* immediately follow letters:

```
grep [^A-Za-z][0-9]+ Homework.java
```

The set `[^A-Za-z]` denotes any characters that are *not* in the ranges A to Z and a to z. This works much better, and it shows only lines that contain actual numbers.

The `useDelimiter` method of the `Scanner` class accepts a regular expression to describe delimiters—the blocks of text that separate words. As already mentioned, if you set the delimiter pattern to `[^A-Za-z]+`, a delimiter is a sequence of one or more characters that are not letters.

There are two useful methods of the `String` class that use regular expressions. The `split` method splits a string into an array of strings, with the delimiter specified as a regular expression. For example,

```
String[] tokens = line.split("\\s+");
```

splits input along white space. The `replaceAll` method yields a string in which all matches of a regular expression are replaced with a string. For example, `word.replaceAll("[aeiou]", "")` is the word with all vowels removed.

For more information on regular expressions, consult one of the many tutorials on the Internet by pointing your search engine to “regular expression tutorial”.

## Special Topic 7.5

**Reading an Entire File**

In the preceding section, you saw how to read lines, words, and characters from a file. Alternatively, you can read the entire file into a list of lines, or into a single string. To do so, use the `Files` class, which provides methods to work with files and directories. The `Files` class requires that you specify file paths as `Path` objects. The `Paths.get` method turns a string containing a file path into such an object. Use these commands:

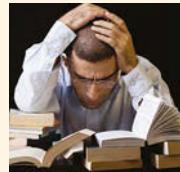
```
String filename = . . .;
List<String> lines = Files.readAllLines(Paths.get(filename));
String content = new String(Files.readAllBytes(Paths.get(filename)));
```



## VIDEO EXAMPLE 7.1

## Computing a Document's Readability

In this Video Example, we develop a program that computes the Flesch Readability Index for a document. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 7.1.



© Ozgur Donmaz/iStockphoto

## 7.3 Command Line Arguments

Depending on the operating system and Java development environment used, there are different methods of starting a program—for example, by selecting “Run” in the compilation environment, by clicking on an icon, or by typing the name of the program at the prompt in a command shell window. The latter method is called “invoking the program from the command line”. When you use this method, you must of course type the name of the program, but you can also type in additional information that the program can use. These additional strings are called **command line arguments**. For example, if you start a program with the command line

```
java ProgramClass -v input.dat
```

then the program receives two command line arguments: the strings “-v” and “input.dat”. It is entirely up to the program what to do with these strings. It is customary to interpret strings starting with a hyphen (-) as program options.

Should you support command line arguments for your programs, or should you prompt users, perhaps with a graphical user interface? For a casual and infrequent user, an interactive user interface is much better. The user interface guides the user along and makes it possible to navigate the application without much knowledge. But for a frequent user, a command line interface has a major advantage: it is easy to automate. If you need to process hundreds of files every day, you could spend all your time typing file names into file chooser dialog boxes. However, by using batch files or shell scripts (a feature of your computer’s operating system), you can automatically call a program many times with different command line arguments.

Your program receives its command line arguments in the args parameter of the main method:

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

In our example, args is an array of length 2, containing the strings

```
args[0]: "-v"
args[1]: "input.dat"
```

Let us write a program that *encrypts* a file—that is, scrambles it so that it is unreadable except to those who know the decryption method. Ignoring 2,000 years of progress in the field of encryption, we will use a method familiar to Julius Caesar, replacing A with a D, B with an E, and so on (see Figure 1).

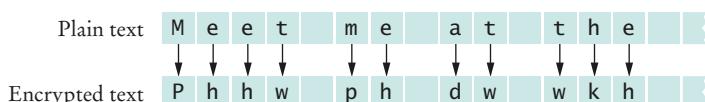


Figure 1 Caesar Cipher

Programs that start from the command line receive the command line arguments in the main method.

© xylo/Stockphoto.



The emperor Julius Caesar used a simple scheme to encrypt messages.

The program takes the following command line arguments:

- An optional -d flag to indicate decryption instead of encryption
- The input file name
- The output file name

For example,

```
java CaesarCipher input.txt encrypt.txt
```

encrypts the file `input.txt` and places the result into `encrypt.txt`.

```
java CaesarCipher -d encrypt.txt output.txt
```

decrypts the file `encrypt.txt` and places the result into `output.txt`.

### sec03/CaesarCipher.java

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.PrintWriter;
4 import java.util.Scanner;
5
6 /**
7 This program encrypts a file using the Caesar cipher.
8 */
9 public class CaesarCipher
10 {
11 public static void main(String[] args) throws FileNotFoundException
12 {
13 final int DEFAULT_KEY = 3;
14 int key = DEFAULT_KEY;
15 String inFile = "";
16 String outFile = "";
17 int files = 0; // Number of command line arguments that are files
18
19 for (int i = 0; i < args.length; i++)
20 {
21 String arg = args[i];
22 if (arg.charAt(0) == '-')
23 {
24 // It is a command line option
25
26 char option = arg.charAt(1);
27 if (option == 'd') { key = -key; }
28 else { usage(); return; }
29 }
30 else
31 {
32 // It is a file name
33
34 files++;
35 if (files == 1) { inFile = arg; }
36 else if (files == 2) { outFile = arg; }
37 }
38 }
39 if (files != 2) { usage(); return; }
40
41 Scanner in = new Scanner(new File(inFile));

```

```

42 in.useDelimiter(""); // Process individual characters
43 PrintWriter out = new PrintWriter(outFile);
44
45 while (in.hasNext())
46 {
47 char from = in.next().charAt(0);
48 char to = encrypt(from, key);
49 out.print(to);
50 }
51 in.close();
52 out.close();
53 }
54
55 /**
56 * Encrypts upper- and lowercase characters by shifting them
57 * according to a key.
58 * @param ch the letter to be encrypted
59 * @param key the encryption key
60 * @return the encrypted letter
61 */
62 public static char encrypt(char ch, int key)
63 {
64 int base = 0;
65 if ('A' <= ch && ch <= 'Z') { base = 'A'; }
66 else if ('a' <= ch && ch <= 'z') { base = 'a'; }
67 else { return ch; } // Not a letter
68 int offset = ch - base + key;
69 final int LETTERS = 26; // Number of letters in the Roman alphabet
70 if (offset >= LETTERS) { offset = offset - LETTERS; }
71 else if (offset < 0) { offset = offset + LETTERS; }
72 return (char) (base + offset);
73 }
74
75 /**
76 * Prints a message describing proper usage.
77 */
78 public static void usage()
79 {
80 System.out.println("Usage: java CaesarCipher [-d] infile outfile");
81 }
82 }
```

**SELF CHECK**

11. If the program is invoked with `java CaesarCipher -d file1.txt`, what are the elements of args?
12. Trace the program when it is invoked as in Self Check 11.
13. Will the program run correctly if the program is invoked with `java CaesarCipher file1.txt file2.txt -d`? If so, why? If not, why not?
14. Encrypt CAESAR using the Caesar cipher.
15. How can you modify the program so that the user can specify an encryption key other than 3 with a -k option? For example,  
`java CaesarCipher -k15 input.txt output.txt`

**Practice It** Now you can try these exercises at the end of the chapter: R7.5, E7.10, E7.11.

**HOW TO 7.1****Processing Text Files**

Processing text files that contain real data can be surprisingly challenging. This How To gives you step-by-step guidance.

**Problem Statement** Read two country data files, `worldpop.txt` and `worldarea.txt` (supplied with the book's companion code). Both files contain the same countries in the same order. Write a file `world_pop_density.txt` that contains country names and population densities (people per square km), with the country names aligned left and the numbers aligned right:

|                |        |
|----------------|--------|
| Afghanistan    | 50.56  |
| Akrotiri       | 127.64 |
| Albania        | 125.91 |
| Algeria        | 14.18  |
| American Samoa | 288.92 |
| ...            |        |



© Oksana Perkins/iStockphoto.

*Singapore is one of the most densely populated countries in the world.*

**Step 1** Understand the processing task.

As always, you need to have a clear understanding of the task before designing a solution. Can you carry out the task by hand (perhaps with smaller input files)? If not, get more information about the problem.

One important aspect that you need to consider is whether you can process the data as it becomes available, or whether you need to store it first. For example, if you are asked to write out sorted data, you first need to collect all input, perhaps by placing it in an array list. However, it is often possible to process the data “on the go”, without storing it.

In our example, we can read each file a line at a time and compute the density for each line because our input files store the population and area data in the same order.

The following pseudocode describes our processing task.

```

While there are more lines to be read
 Read a line from each file.
 Extract the country name.
 population = number following the country name in the line from the first file
 area = number following the country name in the line from the second file
 If area != 0
 density = population / area
 Print country name and density.

```

**Step 2** Determine which files you need to read and write.

This should be clear from the problem. In our example, there are two input files, the population data and the area data, and one output file.

**Step 3** Choose a mechanism for obtaining the file names.

There are three options:

- Hard-coding the file names (such as "`worldpop.txt`").
- Asking the user:

```

Scanner in = new Scanner(System.in);
System.out.print("Enter filename: ");
String inFile = in.nextLine();

```

- Using command-line arguments for the file names.

In our example, we use hard-coded file names for simplicity.

**Step 4** Choose between line, word, and character-based input.

As a rule of thumb, read lines if the input data is grouped by lines. That is the case with tabular data, such as in our example, or when you need to report line numbers.

When gathering data that can be distributed over several lines, then it makes more sense to read words. Keep in mind that you lose all white space when you read words.

Reading characters is mostly useful for tasks that require access to individual characters. Examples include analyzing character frequencies, changing tabs to spaces, or encryption.

**Step 5** With line-oriented input, extract the required data.

It is simple to read a line of input with the `nextLine` method. Then you need to get the data out of that line. You can extract substrings, as described in Section 7.2.4.

Typically, you will use methods such as `Character.isWhitespace` and `Character.isDigit` to find the boundaries of substrings.

If you need any of the substrings as numbers, you must convert them, using `Integer.parseInt` or `Double.parseDouble`.

**Step 6** Use methods to factor out common tasks.

Processing input files usually has repetitive tasks, such as skipping over white space or extracting numbers from strings. It really pays off to develop a set of methods to handle these tedious operations.

In our example, we have two common tasks that call for helper methods: extracting the country name and the value that follows. We will implement methods

```
public static String extractCountry(String line)
public static double extractValue(String line)
```

These methods are implemented as described in Section 7.2.4.

Here is the complete source code (`how_to_1/PopulationDensity.java`).

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
 * This program reads data files of country populations and areas and prints the
 * population density for each country.
 */
public class PopulationDensity
{
 public static void main(String[] args) throws FileNotFoundException
 {
 // Construct Scanner objects for input files

 Scanner in1 = new Scanner(new File("worldpop.txt"));
 Scanner in2 = new Scanner(new File("worldarea.txt"));

 // Construct PrintWriter for the output file

 PrintWriter out = new PrintWriter("world_pop_density.txt");

 // Read lines from each file

 while (in1.hasNextLine() && in2.hasNextLine())
 {
 String line1 = in1.nextLine();
 String line2 = in2.nextLine();
 }
 }
}
```

```

// Extract country and associated value
String country = extractCountry(line1);
double population = extractValue(line1);
double area = extractValue(line2);

// Compute and print the population density
double density = 0;
if (area != 0) // Protect against division by zero
{
 density = population / area;
}
out.printf("%-40s%15.2f%n", country, density);
}

in1.close();
in2.close();
out.close();
}

/**
 * Extracts the country from an input line.
 * @param line a line containing a country name, followed by a number
 * @return the country name
 */
public static String extractCountry(String line)
{
 int i = 0; // Locate the start of the first digit
 while (!Character.isDigit(line.charAt(i))) { i++; }
 return line.substring(0, i).trim(); // Extract the country name
}

/**
 * Extracts the value from an input line.
 * @param line a line containing a country name, followed by a value
 * @return the value associated with the country
 */
public static double extractValue(String line)
{
 int i = 0; // Locate the start of the first digit
 while (!Character.isDigit(line.charAt(i))) { i++; }
 // Extract and convert the value
 return Double.parseDouble(line.substring(i).trim());
}

```



### WORKED EXAMPLE 7.1

### Analyzing Baby Names



In this Worked Example, you will use data from the Social Security Administration to analyze the most popular baby names. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 7.1.



© Nancy Ross/  
iStockphoto.



## Computing & Society 7.1 Encryption Algorithms

This chapter's exercise section gives a few algorithms for encrypting text. Don't actually use any of those methods to send secret messages to your lover. Any skilled cryptographer can *break* these schemes in a very short time—that is, reconstruct the original text without knowing the secret keyword.

In 1978, Ron Rivest, Adi Shamir, and Leonard Adleman introduced an encryption method that is much more powerful. The method is called *RSA encryption*, after the last names of its inventors. The exact scheme is too complicated to present here, but it is not actually difficult to follow. You can find the details in <http://people.csail.mit.edu/rivest/Rsapaper.pdf>.

RSA is a remarkable encryption method. There are two keys: a public key and a private key (see the figure). You can print the public key on your business card (or in your e-mail signature block) and give it to anyone. Then anyone can send you messages that only you can decrypt. Even though everyone else knows the public key, and even if they intercept all the messages coming to you, they cannot break the scheme and actually read the messages. In 1994, hundreds of researchers, collaborating over the Internet, cracked an RSA message encrypted with a 129-digit key. Messages encrypted with a key of 230 digits or more are expected to be secure.

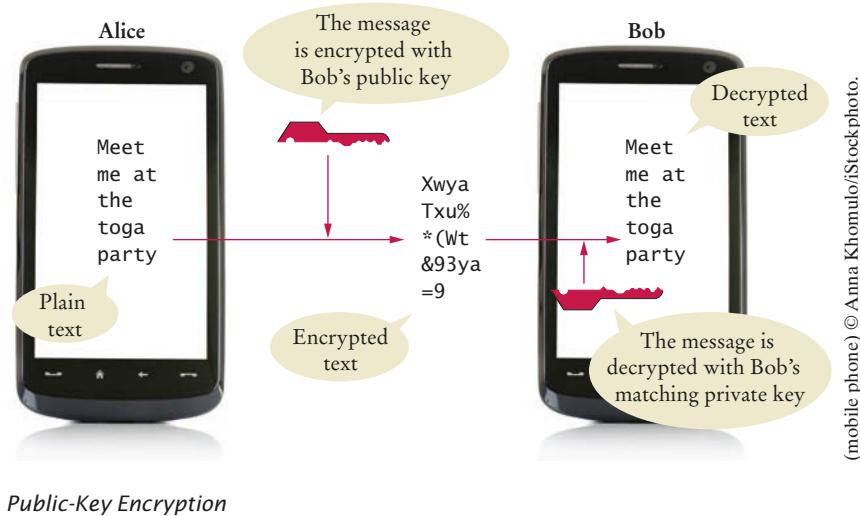
The inventors of the algorithm obtained a *patent* for it. A patent is a deal that society makes with an inventor. For a period of 20 years, the inventor has an exclusive right to its commercialization, may collect royalties from others wishing to manufacture the invention, and may even stop competitors from using it altogether. In return, the inventor must publish the invention, so that others may learn from it, and must relinquish all claim to it after the monopoly period ends. The presumption is that in the absence

of patent law, inventors would be reluctant to go through the trouble of inventing, or they would try to cloak their techniques to prevent others from copying their devices.

There has been some controversy about the RSA patent. Had there not been patent protection, would the inventors have published the method anyway, thereby giving the benefit to society without the cost of the 20-year monopoly? In this case, the answer is probably yes. The inventors were academic researchers who live on salaries rather than sales receipts and are usually rewarded for their discoveries by a boost in their reputation and careers. Would their followers have been as active in discovering (and patenting) improvements? There is no way of knowing, of course. Is an algorithm even patentable, or is it a mathematical fact that belongs to nobody? The patent office did take the latter attitude for a long time. The RSA inventors and many others described their inventions in terms of imaginary electronic devices, rather than algorithms, to circumvent

that restriction. Nowadays, the patent office will award software patents.

There is another interesting aspect to the RSA story. A programmer, Phil Zimmermann, developed a program called PGP (for *Pretty Good Privacy*) that is based on RSA. Anyone can use the program to encrypt messages, and decryption is not feasible even with the most powerful computers. You can get a copy of a free PGP implementation from the GNU project (<http://www.gnupg.org>). The existence of strong encryption methods bothers the United States government to no end. Criminals and foreign agents can send communications that the police and intelligence agencies cannot decipher. The government considered charging Zimmermann with breaching a law that forbids the unauthorized export of munitions, arguing that he should have known that his program would appear on the Internet. There have been serious proposals to make it illegal for private citizens to use these encryption methods, or to keep the keys secret from law enforcement.



## 7.4 Exception Handling

There are two aspects to dealing with program errors: *detection* and *handling*. For example, the Scanner constructor can detect an attempt to read from a non-existent file. However, it cannot handle that error. A satisfactory way of handling the error might be to terminate the program, or to ask the user for another file name. The Scanner class cannot choose between these alternatives. It needs to report the error to another part of the program.

In Java, *exception handling* provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error. In the following sections, we will look into the details of this mechanism.

### 7.4.1 Throwing Exceptions

To signal an exceptional condition, use the `throw` statement to throw an exception object.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program that demonstrates throwing an exception.

When you detect an error condition, your job is really easy. You just *throw* an appropriate exception object, and you are done. For example, suppose someone tries to withdraw too much money from a bank account.

```
if (amount > balance)
{
 // Now what?
}
```

First look for an appropriate exception class. The Java library provides many classes to signal all sorts of exceptional conditions. Figure 2 shows the most useful ones. (The classes are arranged as a tree-shaped hierarchy, with more specialized classes at the bottom of the tree. We will discuss such hierarchies in more detail in Chapter 9.)

Look around for an exception type that might describe your situation. How about the `ArithmaticException`? Is it an arithmetic error to have a negative balance? No—Java can deal with negative numbers. Is the amount to be withdrawn illegal? Indeed it is. It is just too large. Therefore, let's throw an `IllegalArgumentExeption`.

```
if (amount > balance)
{
 throw new IllegalArgumentExeption("Amount exceeds balance");
}
```

### Syntax 7.1 Throwing an Exception

**Syntax**    `throw exceptionObject;`

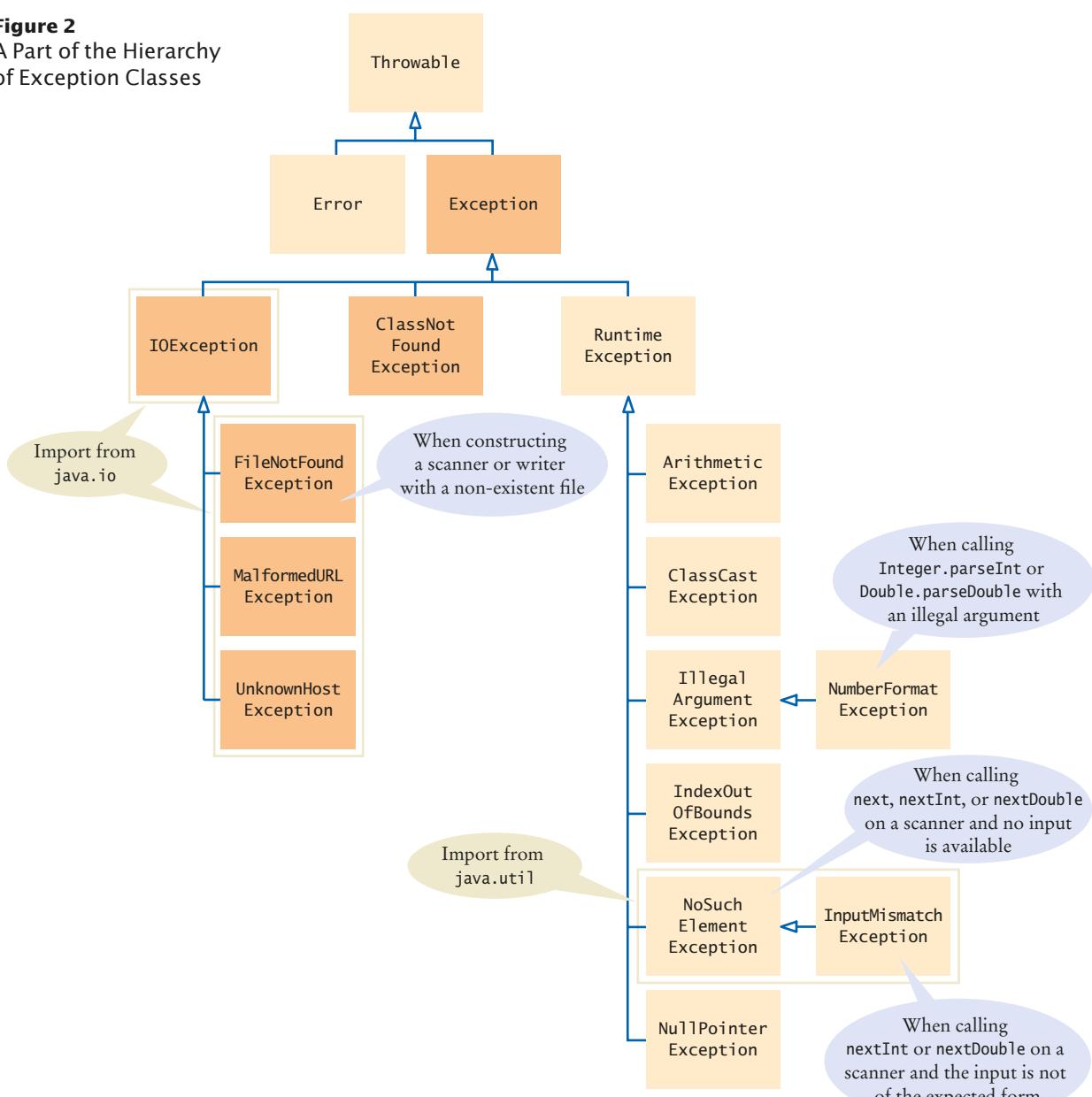
A new exception object is constructed, then thrown.

```
if (amount > balance)
{
 throw new IllegalArgumentExeption("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

**Figure 2**  
A Part of the Hierarchy  
of Exception Classes



When you throw an exception, processing continues in an exception handler.

When you throw an exception, execution does not continue with the next statement but with an **exception handler**. That is the topic of the next section.

*When you throw an exception, the normal control flow is terminated. This is similar to a circuit breaker that cuts off the flow of electricity in a dangerous situation.*



## 7.4.2 Catching Exceptions

Place the statements that can cause an exception inside a try block, and the handler inside a catch clause.

Every exception should be handled somewhere in your program. If an exception has no handler, an error message is printed, and your program terminates. Of course, such an unhandled exception is confusing to program users.

You handle exceptions with the try/catch statement. Place the statement into a location of your program that knows how to handle a particular exception. The try block contains one or more statements that may cause an exception of the kind that you are willing to handle. Each catch clause contains the handler for an exception type. Here is an example:

```
try
{
 String filename = . . .;
 Scanner in = new Scanner(new File(filename));
 String input = in.next();
 int value = Integer.parseInt(input);
 .
 .
}
catch (IOException exception)
{
 exception.printStackTrace();
}
catch (NumberFormatException exception)
{
 System.out.println(exception.getMessage());
}
```

### Syntax 7.2 Catching Exceptions

**Syntax**

```
try
{
 statement
 statement
 .
 .
}
catch (ExceptionClass exceptionObject)
{
 statement
 statement
 .
 .
}
```

When an IOException is thrown, execution resumes here.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

```
try
{
 Scanner in = new Scanner(new File("input.txt"));
 String input = in.next();
 process(input);
}
catch (IOException exception)
{
 System.out.println("Could not open input file");
}
catch (Exception except)
{
 System.out.println(except.getMessage());
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

A FileNotFoundException is a special case of an IOException.

Three exceptions may be thrown in this try block:

- The Scanner constructor can throw a `FileNotFoundException`.
- `Scanner.next` can throw a `NoSuchElementException`.
- `Integer.parseInt` can throw a `NumberFormatException`.

If any of these exceptions is actually thrown, then the rest of the instructions in the try block are skipped. Here is what happens for the various exception types:

- If a `FileNotFoundException` is thrown, then the catch clause for the `IOException` is executed. (If you look at Figure 2, you will note that `FileNotFoundException` is a descendant of `IOException`.) If you want to show the user a different message for a `FileNotFoundException`, you must place the catch clause *before* the clause for an `IOException`.
- If a `NumberFormatException` occurs, then the second catch clause is executed.
- A `NoSuchElementException` is *not caught* by any of the catch clauses. The exception remains thrown until it is caught by another try statement.

Each catch clause contains a handler. When the catch (`IOException exception`) block is executed, then some method in the try block has failed with an `IOException` (or one of its descendants).

In this handler, we produce a printout of the chain of method calls that led to the exception, by calling

```
exception.printStackTrace()
```

In the second exception handler, we call `exception.getMessage()` to retrieve the message associated with the exception. When the `parseInt` method throws a `NumberFormatException`, the message contains the string that it was unable to format. When you throw an exception, you can provide your own message string. For example, when you call

```
throw new IllegalArgumentException("Amount exceeds balance");
```

the message of the exception is the string provided in the constructor.

In these sample catch clauses, we merely inform the user of the source of the problem. Often, it is better to give the user another chance to provide a correct input—see Section 7.5 for a solution.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program that demonstrates catching exceptions.



© Andraz Cerar/Stockphoto.

*You should only catch those exceptions that you can handle.*

### 7.4.3 Checked Exceptions

In Java, the exceptions that you can throw and catch fall into three categories.

- Internal errors are reported by descendants of the type `Error`. One example is the `OutOfMemoryError`, which is thrown when all available computer memory has been used up. These are fatal errors that happen rarely, and we will not consider them in this book.
- Descendants of `RuntimeException`, such as `IndexOutOfBoundsException` or `IllegalArgumentException` indicate errors in your code. They are called **unchecked exceptions**.

Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program that demonstrates throwing and catching checked exceptions.

Add a throws clause to a method that can throw a checked exception.

- All other exceptions are **checked exceptions**. These exceptions indicate that something has gone wrong for some external reason beyond your control. In Figure 2, the checked exceptions are shaded in a darker color.

Why have two kinds of exceptions? A checked exception describes a problem that can occur, no matter how careful you are. For example, an `IOException` can be caused by forces beyond your control, such as a disk error or a broken network connection. The compiler takes checked exceptions very seriously and ensures that they are handled. Your program will not compile if you don't indicate how to deal with a checked exception.

The unchecked exceptions, on the other hand, are your fault. The compiler does not check whether you handle an unchecked exception, such as an `IndexOutOfBoundsException`. After all, you should check your index values rather than install a handler for that exception.

If you have a handler for a checked exception in the same method that may throw it, then the compiler is satisfied. For example,

```
try
{
 File inFile = new File(filename);
 Scanner in = new Scanner(inFile); // Throws FileNotFoundException
 . .
}
catch (FileNotFoundException exception) // Exception caught here
{
 . .
}
```

However, it commonly happens that the current method *cannot handle* the exception. In that case, you need to tell the compiler that you are aware of this exception and that you want your method to be terminated when it occurs. You supply a method with a `throws` clause.

```
public static String readData(String filename) throws FileNotFoundException
{
 File inFile = new File(filename);
 Scanner in = new Scanner(inFile);
 . .
}
```

The `throws` clause signals the caller of your method that it may encounter a `FileNotFoundException`. Then the caller needs to make the same decision—handle the exception, or declare that the exception may be thrown.

It sounds somehow irresponsible not to handle an exception when you know that it happened. Actually, the opposite is true. Java provides an exception handling facility so that an exception can be sent to the *appropriate* handler. Some methods detect errors, some methods handle them, and some methods just pass them along. The `throws` clause simply ensures that no exceptions get lost along the way.



© tillsonburg/iStockphoto.

*Just as trucks with large or hazardous loads carry warning signs, the throws clause warns the caller that an exception may occur.*

## Syntax 7.3 The throws Clause

**Syntax**    *modifiers returnType methodName(parameterType parameterName, . . .)*  
*throws ExceptionClass, ExceptionClass, . . .*

```
public static String readData(String filename)
 throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions  
that this method may throw.

You may also list unchecked exceptions.

### 7.4.4 Closing Resources

When you use a resource that must be closed, such as a `PrintWriter`, you need to be careful in the presence of exceptions. Consider this sequence of statements:

```
PrintWriter out = new PrintWriter(filename);
writeData(out);
out.close(); // May never get here
```

Now suppose that one of the methods before the last line throws an exception. Then the call to `close` is never executed! This is a problem—data that was written to the stream may never end up in the file.

The remedy is to use the **try-with-resources statement**. Declare the `PrintWriter` variable in a `try` statement, like this:

```
try (PrintWriter out = new PrintWriter(filename))
{
 writeData(out);
} // out.close() is always called
```

When the `try` block is completed, the `close` method is called on the variable. If no exception has occurred, this happens when the `writeData` method returns. However, if an exception occurs, the `close` method is invoked before the exception is passed to its handler.

The try-with-resources statement ensures that a resource is closed when the statement ends normally or due to an exception.

## Syntax 7.4

### The try-with-resources Statement

**Syntax**    `try (Type1 variable1 = expression1; Type2 variable2 = expression2; . . .)`  
`{`  
 `. . .`  
`}`

This code may  
throw exceptions.

```
try (PrintWriter out = new PrintWriter(filename))
{
 writeData(out);
}
```

Implements the  
AutoCloseable  
interface.

At this point, `out.close()` is called,  
even when an exception occurs.

You can declare multiple variables in a try-with-resources statement, like this:

```
try (Scanner in = new Scanner(inFile); PrintWriter out = new PrintWriter(outFile))
{
 while (in.hasNextLine())
 {
 String input = in.nextLine();
 String result = process(input);
 out.println(result);
 }
} // Both in.close() and out.close() are called here
```

Use the try-with-resources statement whenever you work with a Scanner or PrintWriter to make sure that these resources are closed properly.

More generally, you can declare variables of any class that implements the AutoCloseable interface in a try-with-resources statement. The classes in the Java library that you use for working with files, network connections, and database connections all implement the AutoCloseable interface.



© archives/iStockphoto.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a program that demonstrates closing resources.

*All visitors to a foreign country have to go through passport control, no matter what happened on their trip. Similarly, the try-with-resources statement ensures that a resource is closed, even when an exception has occurred.*



#### SELF CHECK

16. Suppose balance is 100 and amount is 200. What is the value of balance after these statements?

```
if (amount > balance)
{
 throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

17. When depositing an amount into a bank account, we don't have to worry about overdrafts—except when the amount is negative. Write a statement that throws an appropriate exception in that case.

18. Consider the method

```
public static void main(String[] args)
{
 try
 {
 Scanner in = new Scanner(new File("input.txt"));
 int value = in.nextInt();
 System.out.println(value);
 }
 catch (IOException exception)
 {
 System.out.println("Error opening file.");
 }
}
```

Suppose the file with the given file name exists and has no contents. Trace the flow of execution.

19. Why is an `ArrayIndexOutOfBoundsException` not a checked exception?
20. Is there a difference between catching checked and unchecked exceptions?
21. What is wrong with the following code, and how can you fix it?

```
public static void writeAll(String[] lines, String filename)
{
 PrintWriter out = new PrintWriter(filename);
 for (String line : lines)
 {
 out.println(line.toUpperCase());
 }
 out.close();
}
```

**Practice It** Now you can try these exercises at the end of the chapter: R7.8, R7.9, R7.10.

### Programming Tip 7.1



#### Throw Early, Catch Late

When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix. For example, suppose a method expects to read a number from a file, and the file doesn't contain a number. Simply using a zero value would be a poor choice because it hides the actual problem and perhaps causes a different problem elsewhere.

Conversely, a method should only catch an exception if it can really remedy the situation. Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

These principles can be summarized with the slogan “throw early, catch late”.

Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.

### Programming Tip 7.2



#### Do Not Squelch Exceptions

When you call a method that throws a checked exception and you haven't specified a handler, the compiler complains. In your eagerness to continue your work, it is an understandable impulse to shut the compiler up by squelching the exception:

```
try
{
 Scanner in = new Scanner(new File(filename));
 // Compiler complained about FileNotFoundException
 ...
}
catch (FileNotFoundException e) {} // So there!
```

The do-nothing exception handler fools the compiler into thinking that the exception has been handled. In the long run, this is clearly a bad idea. Exceptions were designed to transmit problem reports to a competent handler. Installing an incompetent handler simply hides an error condition that could be serious.

**Programming Tip 7.3****Do Throw Specific Exceptions**

When throwing an exception, you should choose an exception class that describes the situation as closely as possible. For example, it would be a bad idea to simply throw a `RuntimeException` object when a bank account has insufficient funds. This would make it far too difficult to catch the exception. After all, if you caught all exceptions of type `RuntimeException`, your catch clause would also be activated by exceptions of the type `NullPointerException`, `ArrayIndexOutOfBoundsException`, and so on. You would then need to carefully examine the exception object and attempt to deduce whether the exception was caused by insufficient funds.

If the standard library does not have an exception class that describes your particular error situation, simply provide a new exception class.

**Special Topic 7.6****Assertions**

An **assertion** is a condition that you believe to be true at all times in a particular program location. An assertion check tests whether an assertion is true. Here is a typical assertion check:

```
public double deposit(double amount)
{
 assert amount >= 0;
 balance = balance + amount;
}
```

In this method, the programmer expects that the quantity `amount` can never be negative. When the assertion is correct, no harm is done, and the program works in the normal way. If, for some reason, the assertion fails, and assertion checking is enabled, then the `assert` statement throws an exception of type `AssertionError`, causing the program to terminate.

However, if assertion checking is disabled, then the assertion is never checked, and the program runs at full speed. By default, assertion checking is disabled when you execute a program.

To execute a program with assertion checking turned on, use this command:

```
java -enableassertions MainClass
```

You can also use the shortcut `-ea` instead of `-enableassertions`. You should turn assertion checking on during program development and testing.

**Special Topic 7.7****The try/finally Statement**

You saw in Section 7.4.4 how to ensure that a resource is closed when an exception occurs. The `try-with-resources` statement calls the `close` methods of variables declared within the statement header. You should always use the `try-with-resources` statement when closing resources.

It can happen that you need to do some cleanup other than calling a `close` method. In that case, use the `try/finally` statement:

```
public double deposit (double amount)
try
{
 . . .
}
finally
{
 Cleanup. // This code is executed whether or not an exception occurs
}
```

If the body of the `try` statement completes without an exception, the cleanup happens. If an exception is thrown, the cleanup happens and the exception is then propagated to its handler.

The try/finally statement is rarely required because most Java library classes that require cleanup implement the AutoCloseable interface.



## Computing & Society 7.2 The Ariane Rocket Incident

The European Space Agency (ESA), Europe's counterpart to NASA, had developed a rocket model called Ariane that it had successfully used several times to launch satellites and scientific experiments into space. However, when a new version, the Ariane 5, was launched on June 4, 1996, from ESA's launch site in Kourou, French Guiana, the rocket veered off course about 40 seconds after liftoff. Flying at an angle of more than 20 degrees, rather than straight up, exerted such an aerodynamic force that the boosters separated, which triggered the automatic self-destruction mechanism. The rocket blew itself up.

The ultimate cause of this accident was an unhandled exception! The rocket contained two identical devices (called inertial reference systems) that processed flight data from measuring devices and turned the data into information about the rocket position.

The onboard computer used the position information for controlling the boosters. The same inertial reference systems and computer software had worked fine on the Ariane 4.

However, due to design changes to the rocket, one of the sensors measured a larger acceleration force than had been encountered in the Ariane 4. That value, expressed as a floating-point value, was stored in a 16-bit integer (like a short variable in Java). Unlike Java, the Ada language, used for the device software, generates an exception if a floating-point number is too large to be converted to an integer. Unfortunately, the programmers of the device had decided that this situation would never happen and didn't provide an exception handler.

When the overflow did happen, the exception was triggered and, because there was no handler, the device shut itself off. The onboard computer sensed

the failure and switched over to the backup device. However, that device had shut itself off for exactly the same reason, something that the designers of the rocket had not expected. They figured that the devices might fail for mechanical reasons, and the chance of two devices having the same mechanical failure was considered remote. At that point, the rocket was without reliable position information and went off course. Perhaps it would have been better if the software hadn't been so thorough? If it had ignored the overflow, the device wouldn't have been shut off. It would have computed bad data. But then the device would have reported wrong position data, which could have been just as fatal. Instead, a correct implementation should have caught overflow exceptions and come up with some strategy to recompute the flight data. Clearly, giving up was not a reasonable option in this context.

The advantage of the exception-handling mechanism is that it makes these issues explicit to programmers—something to think about when you curse the Java compiler for complaining about uncaught exceptions.

© AP/Wide World Photos.



*The Explosion of the Ariane Rocket*

## 7.5 Application: Handling Input Errors

This section walks through an example program that includes exception handling. The program, `DataAnalyzer.java`, asks the user for the name of a file. The file is expected to contain data values. The first line of the file should contain the total number of values, and the remaining lines contain the data. A typical input file looks like this:

```
3
1.45
-2.1
0.05
```

When designing a program, ask yourself what kinds of exceptions can occur.

For each exception, you need to decide which part of your program can competently handle it.

What can go wrong? There are two principal risks.

- The file might not exist.
- The file might have data in the wrong format.

Who can detect these faults? The Scanner constructor will throw an exception when the file does not exist. The methods that process the input values need to throw an exception when they find an error in the data format.

What exceptions can be thrown? The Scanner constructor throws a `FileNotFoundException` when the file does not exist, which is appropriate in our situation. When there are fewer data items than expected, or when the file doesn't start with the count of values, the program will throw an `NoSuchElementException`. Finally, when there are more inputs than expected, an `IOException` should be thrown.

Who can remedy the faults that the exceptions report? Only the `main` method of the `DataAnalyzer` program interacts with the user, so it catches the exceptions, prints appropriate error messages, and gives the user another chance to enter a correct file:

```
// Keep trying until there are no more exceptions
boolean done = false;
while (!done)
{
 try
 {
 Prompt user for file name.

 double[] data = readFile(filename);

 Process data.

 done = true;
 }
 catch (FileNotFoundException exception)
 {
 System.out.println("File not found.");
 }
 catch (NoSuchElementException exception)
 {
 System.out.println("File contents invalid.");
 }
 catch (IOException exception)
 {
 exception.printStackTrace();
 }
}
```

The first two catch clauses in the `main` method give a human-readable error report if bad data was encountered or the file was not found. However, if another `IOException` occurs, then it prints a stack trace so that a programmer can diagnose the problem.

The following `readFile` method constructs the `Scanner` object and calls the `readData` method. It does not handle any exceptions. If there is a problem with the input file, it simply passes the exception to its caller.

```
public static double[] readFile(String filename) throws IOException
{
 File inFile = new File(filename);
 try (Scanner in = new Scanner(inFile))
 {
 return readData(in);
 }
```

```
 }
}
```

Note how the `try-with-resources` statement ensures that the file is closed even when an exception occurs.

Also note that the `throws` clause of the `readFile` method need not include the `FileNotFoundException` class because it is a special case of an `IOException`.

The `readData` method reads the number of values, constructs an array, and fills it with the data values.

```
public static double[] readData(Scanner in) throws IOException
{
 int numberofValues = in.nextInt(); // May throw NoSuchElementException
 double[] data = new double[numberofValues];

 for (int i = 0; i < numberofValues; i++)
 {
 data[i] = in.nextDouble(); // May throw NoSuchElementException
 }

 if (in.hasNext())
 {
 throw new IOException("End of file expected");
 }
 return data;
}
```

As discussed in Section 7.2.7, the calls to the `nextInt` and `nextDouble` methods can throw a `NoSuchElementException` when there is no input at all or an `InputMismatchException` if the input is not a number. As you can see from Figure 2 on page 340, an `InputMismatchException` is a special case of a `NoSuchElementException`.

You need not declare the `NoSuchElementException` in the `throws` clause because it is not a checked exception, but you can include it for greater clarity.

There are three potential errors:

- The file might not start with an integer.
- There might not be a sufficient number of data values.
- There might be additional input after reading all data values.

In the first two cases, the `Scanner` throws a `NoSuchElementException`. Note again that this is *not* a checked exception—we could have avoided it by calling `hasNextInt`/`hasNextDouble` first. However, this method does not know what to do in this case, so it allows the exception to be sent to a handler elsewhere.

When we find that there is additional unexpected input, we throw an `IOException`. To see the exception handling at work, look at a specific error scenario.

1. `main` calls `readFile`.
2. `readFile` calls `readData`.
3. `readData` calls `Scanner.nextInt`.
4. There is no integer in the input, and `Scanner.nextInt` throws a `NoSuchElementException`.
5. `readData` has no catch clause. It terminates immediately.
6. `readFile` has no catch clause. It terminates immediately when leaving the `try-with-resources` statement.

7. The first catch clause in `main` is for a `FileNotFoundException`. The exception that is currently being thrown is a `NoSuchElementException`, and this handler doesn't apply.
8. The next catch clause is for a `NoSuchElementException`, and execution resumes here. That handler prints a message to the user. Afterward, the user is given another chance to enter a file name. Note that the statements for processing the data have been skipped.

This example shows the separation between error detection (in the `readData` method) and error handling (in the `main` method). In between the two is the `readFile` method, which simply passes the exceptions along.

### sec05/DataAnalyzer.java

```

1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.io.IOException;
4 import java.util.Scanner;
5 import java.util.NoSuchElementException;
6
7 /**
8 This program processes a file containing a count followed by data values.
9 If the file doesn't exist or the format is incorrect, you can specify another file.
10 */
11 public class DataAnalyzer
12 {
13 public static void main(String[] args)
14 {
15 Scanner in = new Scanner(System.in);
16
17 // Keep trying until there are no more exceptions
18
19 boolean done = false;
20 while (!done)
21 {
22 try
23 {
24 System.out.print("Please enter the file name: ");
25 String filename = in.next();
26
27 double[] data = readFile(filename);
28
29 // As an example for processing the data, we compute the sum
30
31 double sum = 0;
32 for (double d : data) { sum = sum + d; }
33 System.out.println("The sum is " + sum);
34
35 done = true;
36 }
37 catch (FileNotFoundException exception)
38 {
39 System.out.println("File not found.");
40 }
41 catch (NoSuchElementException exception)
42 {
43 System.out.println("File contents invalid.");
44 }
45 }
46 }
47 }
```

```

44 }
45 catch (IOException exception)
46 {
47 exception.printStackTrace();
48 }
49 }
50 }
51 /**
52 * Opens a file and reads a data set.
53 * @param filename the name of the file holding the data
54 * @return the data in the file
55 */
56 public static double[] readFile(String filename) throws IOException
57 {
58 File inFile = new File(filename);
59 try (Scanner in = new Scanner(inFile))
60 {
61 return readData(in);
62 }
63 }
64 }
65 /**
66 * Reads a data set.
67 * @param in the scanner that scans the data
68 * @return the data set
69 */
70 public static double[] readData(Scanner in) throws IOException
71 {
72 int numberofValues = in.nextInt(); // May throw NoSuchElementException
73 double[] data = new double[numberofValues];
74
75 for (int i = 0; i < numberofValues; i++)
76 {
77 data[i] = in.nextDouble(); // May throw NoSuchElementException
78 }
79
80 if (in.hasNext())
81 {
82 throw new IOException("End of file expected");
83 }
84 return data;
85 }
86 }
87 }

```

**SELF CHECK**

- 22.** Why doesn't the `readFile` method catch any exceptions?
- 23.** What happens to the `Scanner` object if the `readData` method throws an exception?
- 24.** What happens to the `Scanner` object if the `readData` method doesn't throw an exception?
- 25.** Suppose the user specifies a file that exists and is empty. Trace the flow of execution in the `DataAnalyzer` program.
- 26.** Why didn't the `readData` method call `hasNextInt/hasNextDouble` to ensure that the `NoSuchElementException` is not thrown?

**Practice It** Now you can try these exercises at the end of the chapter: R7.16, R7.17, E7.12.

**VIDEO EXAMPLE 7.2****Detecting Accounting Fraud**

In this Video Example, you will see how to detect accounting fraud by analyzing digit distributions. You will learn how to read data from the Internet and handle exceptional situations. Go to [wiley.com/go/bj102videos](http://wiley.com/go/bj102videos) to view Video Example 7.2.



© Noreenbo/iStockphoto.

**CHAPTER SUMMARY****Develop programs that read and write files.**

- Use the `Scanner` class for reading text files.
- When writing text files, use the `PrintWriter` class and the `print`/`println`/`printf` methods.
- Close all files when you are done processing them.

**Be able to process text in files.**

- The `next` method reads a string that is delimited by white space.
- The `Character` class has methods for classifying characters.
- The `nextLine` method reads an entire line.
- If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.

**Process the command line arguments of a program.**

- Programs that start from the command line receive the command line arguments in the `main` method.

**Use exception handling to transfer control from an error location to an error handler.**

- To signal an exceptional condition, use the `throw` statement to throw an exception object.
- When you throw an exception, processing continues in an exception handler.
- Place the statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause.
- Checked exceptions are due to external circumstances that the programmer cannot prevent. The compiler checks that your program handles these exceptions.
- Add a `throws` clause to a method that can throw a checked exception.
- The `try-with-resources` statement ensures that a resource is closed when the statement ends normally or due to an exception.
- Throw an exception as soon as a problem is detected. Catch it only when the problem can be handled.



**Use exception handling in a program that processes input.**

- When designing a program, ask yourself what kinds of exceptions can occur.
- For each exception, you need to decide which part of your program can competently handle it.

**STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER**

|                                    |                                  |
|------------------------------------|----------------------------------|
| java.io.File                       | java.lang.RuntimeException       |
| java.io.FileNotFoundException      | java.lang.String                 |
| java.io.IOException                | replaceAll                       |
| java.io.PrintWriter                | split                            |
| close                              | java.lang.Throwable              |
| java.lang.AutoCloseable            | getMessage                       |
| java.lang.Character                | printStackTrace                  |
| isDigit                            | java.net.URL                     |
| isLetter                           | openStream                       |
| isLowerCase                        | java.util.InputMismatchException |
| isUpperCase                        | java.util.NoSuchElementException |
| isWhiteSpace                       | java.util.Scanner                |
| java.lang.Double                   | close                            |
| parseDouble                        | hasNextLine                      |
| java.lang.Error                    | nextLine                         |
| java.lang.Integer                  | useDelimiter                     |
| parseInt                           | javax.swing.JFileChooser         |
| java.lang.IllegalArgumentException | getSelectedFile                  |
| java.lang.NullPointerException     | showOpenDialog                   |
| java.lang.NumberFormatException    | showSaveDialog                   |

**REVIEW EXERCISES**

- **R7.1** What happens if you try to open a file for reading that doesn't exist? What happens if you try to open a file for writing that doesn't exist?
- **R7.2** What happens if you try to open a file for writing, but the file or device is write-protected (sometimes called read-only)? Try it out with a short test program.
- **R7.3** What happens when you write to a PrintWriter without closing it? Produce a test program that demonstrates how you can lose data.
- **R7.4** How do you open a file whose name contains a backslash, like c:\temp\output.dat?
- **R7.5** If a program Woozle is started with the command  

```
java Woozle -Dname=piglet -I\eyore -v heff.txt a.txt lump.txt
```

what are the values of args[0], args[1], and so on?
- **R7.6** What is the difference between throwing an exception and catching an exception?
- **R7.7** What is a checked exception? What is an unchecked exception? Give an example for each. Which exceptions do you need to declare with the throws reserved word?
- **R7.8** Why don't you need to declare that your method might throw an IndexOutOfBoundsException?

- R7.9 When your program executes a `throw` statement, which statement is executed next?
- R7.10 What happens if an exception does not have a matching `catch` clause?
- R7.11 What can your program do with the exception object that a `catch` clause receives?
- R7.12 Is the type of the exception object always the same as the type declared in the `catch` clause that catches it? If not, why not?
- R7.13 What is the purpose of the `try-with-resources` statement? Give an example of how it can be used.
- R7.14 What happens when an exception is thrown, a `try-with-resources` statement calls `close`, and that call throws an exception of a different kind than the original one? Which one is caught by a surrounding `catch` clause? Write a sample program to try it out.
- R7.15 Which exceptions can the `next` and `nextInt` methods of the `Scanner` class throw? Are they checked exceptions or unchecked exceptions?
- R7.16 Suppose the program in Section 7.5 reads a file containing the following values:

```

1
2
3
4

```

What is the outcome? How could the program be improved to give a more accurate error report?

- R7.17 Can the `readFile` method in Section 7.5 throw a `NullPointerException`? If so, how?
- R7.18 The following code tries to close the writer without using a `try-with-resources` statement:

```

PrintWriter out = new PrintWriter(filename);
try
{
 Write output.
 out.close();
}
catch (IOException exception)
{
 out.close();
}

```

What is the disadvantage of this approach? (*Hint:* What happens when the `PrintWriter` constructor or the `close` method throws an exception?)

## PRACTICE EXERCISES

- E7.1 Write a program that carries out the following tasks:

- Open a file with the name `hello.txt`.**
- Store the message "Hello, World!" in the file.**
- Close the file.**
- Open the same file again.**
- Read the message into a string variable and print it.**

- **E7.2** Write a program that reads a file, removes any blank lines, and writes the non-blank lines back to the same file.
- **E7.3** Write a program that reads a file, removes any blank lines at the beginning or end of the file, and writes the remaining lines back to the same file.
- **E7.4** Write a program that reads a file containing text. Read each line and send it to the output file, preceded by *line numbers*. If the input file is

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

then the program produces the output file

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```



© Chris Price/Stockphoto

The line numbers are enclosed in /\* \*/ delimiters so that the program can be used for numbering Java source files.

Prompt the user for the input and output file names.

- **E7.5** Repeat Exercise E7.4, but allow the user to specify the file name on the command-line. If the user doesn't specify any file name, then prompt the user for the name.
- **E7.6** Write a program that reads a file containing two columns of floating-point numbers. Prompt the user for the file name. Print the average of each column.
- **E7.7** Write a program that asks the user for a file name and prints the number of characters, words, and lines in that file.
- **E7.8** Write a program `Find` that searches all files specified on the command line and prints out all lines containing a specified word. For example, if you call

```
java Find ring report.txt address.txt Homework.java
```

then the program might print

```
report.txt: has broken up an international ring of DVD bootleggers that
address.txt: Kris Kringle, North Pole
address.txt: Homer Simpson, Springfield
Homework.java: String filename;
```

The specified word is always the first command line argument.

- **E7.9** Write a program that checks the spelling of all words in a file. It should read each word of a file and check whether it is contained in a word list. A word list is available on most Linux systems in the file /usr/share/dict/words. (If you don't have access to a Linux system, your instructor should be able to get you a copy.) The program should print out all words that it cannot find in the word list.
- **E7.10** Write a program that replaces each line of a file with its reverse. For example, if you run

```
java Reverse HelloPrinter.java
```

then the contents of `HelloPrinter.java` are changed to

```
retnirPolleH ssalc cilbup
{
)sgra]gnirtS(niam diov citats cilbup
```

```
{
wodniw elosnoc eht ni gniteerg a yalpsiD //

;)"!dlroW ,olleH"(nltinirp.tuo.metsyS

}

}
```

Of course, if you run Reverse twice on the same file, you get back the original file.

- E7.11 Write a program that reads each line in a file, reverses its lines, and writes them to another file. For example, if the file `input.txt` contains the lines

```
Mary had a little lamb

Its fleece was white as snow

And everywhere that Mary went

The lamb was sure to go.
```

and you run

```
reverse input.txt output.txt
```

then `output.txt` contains

```
The lamb was sure to go.

And everywhere that Mary went

Its fleece was white as snow

Mary had a little lamb
```

- E7.12 Write a program that asks the user to input a set of floating-point values. When the user enters a value that is not a number, give the user a second chance to enter the value. After two chances, quit reading input. Add all correctly specified values and print the sum when the user is done entering data. Use exception handling to detect improper inputs.
- E7.13 Modify the `DataAnalyzer` program so that you do not call `hasNextInt` or `hasNextDouble`. Simply have `nextInt` and `nextDouble` throw a `NoSuchElementException` and catch it in the `main` method.

## PROGRAMMING PROJECTS

- P7.1 Get the data for names in prior decades from the Social Security Administration. Paste the table data in files named `babynames80s.txt`, etc. Modify the `worked_example_1/BabyNames.java` program so that it prompts the user for a file name. The numbers in the files have comma separators, so modify the program to handle them. Can you spot a trend in the frequencies?
- P7.2 Write a program that reads in `worked_example_1/babynames.txt` and produces two files, `boynames.txt` and `girlnames.txt`, separating the data for the boys and girls.
- P7.3 Write a program that reads a file in the same format as `worked_example_1/babynames.txt` and prints all names that are both boy and girl names (such as Alexis or Morgan).
- P7.4 Using the mechanism described in Special Topic 7.1, write a program that reads all data from a web page and writes them to a file. Prompt the user for the web page URL and the file.
- P7.5 The CSV (or *comma-separated values*) format is commonly used for tabular data. Each table row is a line, with columns separated by commas. Items may be enclosed

in quotation marks, and they must be if they contain commas or quotation marks. Quotation marks inside quoted fields are doubled. Here is a line with four fields:

1729, San Francisco, "Hello, World", "He asked: ""Quo vadis?"""

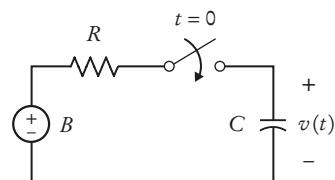
Write a CSVReader program that reads a CSV file, and provide methods

```
int numberOfRows()
int numberOfFields(int row)
String field(int row, int column)
```

- ■ **P7.6** Find an interesting data set in CSV format (or in spreadsheet format, then use a spreadsheet to save the data as CSV). Using the CSVReader program from Exercise P7.5, read the data and compute a summary, such as the maximum, minimum, or average of one of the columns.
- ■ **P7.7** Using the mechanism described in Special Topic 7.1, write a program that reads all data from a web page and prints all hyperlinks of the form  

$$<\text{a href}=\text{"link"}>\text{link text}</\text{a}>$$
 Extra credit if your program can follow the links that it finds and find links in those web pages as well. (This is the method that search engines such as Google use to find web sites.)
- ■ **Business P7.8** A hotel salesperson enters sales in a text file. Each line contains the following, separated by semicolons: The name of the client, the service sold (such as Dinner, Conference, Lodging, and so on), the amount of the sale, and the date of that event. Write a program that reads such a file and displays the total amount for each service category. Display an error if the file does not exist or the format is incorrect.
- ■ **Business P7.9** Write a program that reads a text file, as described in Exercise P7.8, and writes a separate file for each service category. Each service category file should contain the entries for that category. Name the output files Dinner.txt, Conference.txt, and so on.
- ■ **Business P7.10** A store owner keeps a record of daily cash transactions in a text file. Each line contains three items: The invoice number, the cash amount, and the letter P if the amount was paid or R if it was received. Items are separated by spaces. Write a program that prompts the store owner for the amount of cash at the beginning and end of the day, and the name of the file. Your program should check whether the actual amount of cash at the end of the day equals the expected value.
- ■ ■ **Science P7.11** After the switch in the figure below closes, the voltage (in volts) across the capacitor is represented by the equation

$$v(t) = B \left( 1 - e^{-t/(RC)} \right)$$



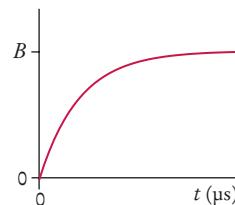
Suppose the parameters of the electric circuit are  $B = 12$  volts,  $R = 500 \Omega$ , and  $C = 0.25 \mu\text{F}$ . Consequently

$$v(t) = 12(1 - e^{-0.008t})$$

where  $t$  has units of  $\mu\text{s}$ . Read a file `params.txt` containing the values for  $B$ ,  $R$ ,  $C$ , and the starting and ending values for  $t$ . Write a file `rc.txt` of values for the time  $t$  and the corresponding capacitor voltage  $v(t)$ , where  $t$  goes from the given starting value to the given ending value in 100 steps. In our example, if  $t$  goes from 0 to 1,000  $\mu\text{s}$ , the twelfth entry in the output file would be:

110 7.02261

- Science P7.12** The figure below shows a plot of the capacitor voltage from the circuit shown in Exercise P7.11. The capacitor voltage increases from 0 volts to  $B$  volts. The “rise time” is defined as the time required for the capacitor voltage to change from  $v_1 = 0.05 \times B$  to  $v_2 = 0.95 \times B$ .



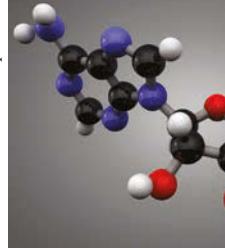
The file `rc.txt` contains a list of values of time  $t$  and the corresponding capacitor voltage  $v(t)$ . A time in  $\mu\text{s}$  and the corresponding voltage in volts are printed on the same line. For example, the line

110 7.02261

indicates that the capacitor voltage is 7.02261 volts when the time is 110  $\mu\text{s}$ . The time is increasing in the data file.

Write a program that reads the file `rc.txt` and uses the data to calculate the rise time. Approximate  $B$  by the voltage in the last line of the file, and find the data points that are closest to  $0.05 \times B$  and  $0.95 \times B$ .

- Science P7.13** Suppose a file contains bond energies and bond lengths for covalent bonds in the following format:



| Single, double, or triple bond | Bond energy (kJ/mol) | Bond length (nm) |
|--------------------------------|----------------------|------------------|
| C C                            | 370                  | 0.154            |
| C  C                           | 680                  | 0.13             |
| C  C                           | 890                  | 0.12             |
| C H                            | 435                  | 0.11             |
| C N                            | 305                  | 0.15             |

| Single, double,<br>or triple bond | Bond energy<br>(kJ/mol) | Bond length<br>(nm) |
|-----------------------------------|-------------------------|---------------------|
| C O                               | 360                     | 0.14                |
| C F                               | 450                     | 0.14                |
| C Cl                              | 340                     | 0.18                |
| O H                               | 500                     | 0.10                |
| O O                               | 220                     | 0.15                |
| O Si                              | 375                     | 0.16                |
| N H                               | 430                     | 0.10                |
| N O                               | 250                     | 0.12                |
| F F                               | 160                     | 0.14                |
| H H                               | 435                     | 0.074               |

Write a program that accepts data from one column and returns the corresponding data from the other columns in the stored file. If input data matches different rows, then return all matching row data. For example, a bond length input of 0.12 should return triple bond C||C and bond energy 890 kJ/mol *and* single bond N|O and bond energy 250 kJ/mol.

## ANSWERS TO SELF-CHECK QUESTIONS

- When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the `while` loop exits immediately.
- The program throws a `FileNotFoundException` and terminates.
- Open a scanner for the file.**  
**For each number in the scanner**  
**Add the number to an array.**  
**Close the scanner.**  
**Set total to 0.**  
**Open a print writer for the file.**  
**For each number in the array**  
**Write the number to the print writer.**  
**Add the number to total.**  
**Write total to the print writer.**  
**Close the print writer.**
- Add a variable `count` that is incremented whenever a number is read. In the end, print the average, not the total, as  
`out.printf("Average: %8.2f%n", total / count);`  
Because the string "Average" is three characters longer than "Total", change the other output to  
`out.printf("%18.2f%n", value).`
- Add a variable `count` that is incremented whenever a number is read. Only write a new line when it is even.  
`count++;  
out.printf("%8.2f", value);  
if (count % 2 == 0) { out.println(); }`  
At the end of the loop, write a new line if `count` is odd, then write the total:  
`if (count % 2 == 1) { out.println(); }  
out.printf("Total: %10.2f%n", total);`

6. word is "Hello," and input is "World!"
7. Because 995.0 is not an integer, the call `in.hasNextInt()` returns false, and the call `in.nextInt()` is skipped. The value of number stays 0, and input is set to the string "995.0".
8. `x1` is set to 6000000. Because a dollar sign is not considered a part of a floating-point number in Java, the second call to `nextDouble` causes an input mismatch exception and `x2` is not set.
9. Read them as strings, and convert those strings to numbers that are not equal to N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
 double value = Double.parseDouble(input);
 Process value.
}
```

10. Locate the last character of the country name:

```
int j = i - 1;
while (!Character.isWhiteSpace(line.charAt(j)))
{
 j--;
}
```

Then extract the country name:

```
String countryName = line.substring(0, j + 1);
```

11. `args[0]` is "-d" and `args[1]` is "file1.txt"

| key | inFile    | outFile | i | arg       |
|-----|-----------|---------|---|-----------|
| 3   | null      | null    | 0 | -d        |
| -3  | file1.txt |         | 1 | file1.txt |
|     |           |         | 2 |           |

Then the program prints a message

Usage: java CaesarCipher [-d] infile outfile

13. The program will run correctly. The loop that parses the options does not depend on the positions in which the options appear.

14. FDHVDU

15. Add the lines

```
else if (option == 'k')
{
 key = Integer.parseInt(
 args[i].substring(2));
}
```

after line 27 and update the usage information.

16. It is still 100. The last statement was not executed because the exception was thrown.
17. 

```
if (amount < 0)
{
 throw new IllegalArgumentException(
 "Negative amount");
}
```
18. The Scanner constructor succeeds because the file exists. The `nextInt` method throws a `NoSuchElementException`. This is *not* an `IOException`. Therefore, the error is not caught. Because there is no other handler, an error message is printed and the program terminates.
19. Because programmers should simply check that their array index values are valid instead of trying to handle an `ArrayIndexOutOfBoundsException`.
20. No. You can catch both exception types in the same way, as you can see in the code example on page 354.
21. There are two mistakes. The `PrintWriter` constructor can throw a `FileNotFoundException`. You should supply a `throws` clause. And if one of the array elements is `null`, a `NullPointerException` is thrown. In that case, the `out.close()` statement is never executed. You should use a `try-with-resources` statement.
22. The exceptions are better handled in the `main` method.
23. The `close` method is called on the `Scanner` object before the exception is propagated to its handler.
24. The `close` method is called on the `Scanner` object before the `readFile` method returns to its caller.
25. `main` calls `readFile`, which calls `readData`. The call `in.nextInt()` throws a `NoSuchElementException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught. An error message is printed, and the user can specify another file.
26. We *want* to throw that exception, so that someone else can handle the problem of a bad data file.

## WORKED EXAMPLE 7.1

## Analyzing Baby Names



**Problem Statement** The Social Security Administration publishes lists of the most popular baby names on their web site <http://www.ssa.gov/OACT/babynames/>. When querying the 1,000 most popular names for a given decade, the browser displays the result on the screen (see the *Querying Baby Names* figure below).

To save the data as text, one simply selects it and pastes the result into a file. The book code contains a file `babynames.txt` with the data for the 1990s.

Each line in the file contains seven entries:

- The rank (from 1 to 1,000)
- The name, frequency, and percentage of the male name of that rank
- The name, frequency, and percentage of the female name of that rank

For example, the line

```
10 Joseph 260365 1.2681 Megan 160312 0.8168
```

shows that the 10th most common boy's name was Joseph, with 260,365 births, or 1.2681 percent of all births during that period. The 10th most common girl's name was Megan. Why are there many more Josephs than Megans? Parents seem to use a wider set of girl's names, making each one of them less frequent.

Your task is to test that conjecture, by determining the names given to the top 50 percent of boys and girls in the list. Simply print boy and girl names, together with their ranks, until you reach the 50 percent limit.



© Nancy Ross/Stockphoto.

### Step 1 Understand the processing task.

To process each line, we first read the rank. We then read three values (name, count, and percentage) for the boy's name. Then we repeat that step for girls. To stop processing after reaching 50 percent, we can add up the frequencies and stop when they reach 50 percent.

We need separate totals for boys and girls. When a total reaches 50 percent, we stop printing. When both totals reach 50 percent, we stop reading.

The following pseudocode describes our processing task.

```
boyTotal = 0
girlTotal = 0
While boyTotal < 50 or girlTotal < 50
 Read the rank and print it.
 Read the boy name, count, and percentage.
 If boyTotal < 50
 Print boy name.
 Add percentage to boyTotal.
 Repeat for girl part.
```

### Step 2 Determine which files you need to read and write.

We only need to read a single file, `babynames.txt`. We were not asked to save the output to a file, so we will just send it to `System.out`.

### Step 3 Choose a mechanism for obtaining the file names.

We do not need to prompt the user for the file name.

| Rank | Name        | Male    |         | Female    |         |         |
|------|-------------|---------|---------|-----------|---------|---------|
|      |             | Number  | Percent | Name      | Number  | Percent |
| 1    | Michael     | 462,085 | 2.2506  | Jessica   | 302,962 | 1.5436  |
| 2    | Christopher | 361,250 | 1.7595  | Ashley    | 301,702 | 1.5372  |
| 3    | Matthew     | 351,477 | 1.7119  | Emily     | 237,133 | 1.2082  |
| 4    | Joshua      | 328,955 | 1.6022  | Sarah     | 224,000 | 1.1413  |
| 5    | Jacob       | 298,016 | 1.4515  | Samantha  | 223,913 | 1.1408  |
| 6    | Nicholas    | 275,222 | 1.3405  | Amanda    | 190,901 | 0.9726  |
| 7    | Andrew      | 272,600 | 1.3277  | Brittany  | 190,779 | 0.9720  |
| 8    | Daniel      | 271,734 | 1.3235  | Elizabeth | 172,383 | 0.8783  |
| 9    | Tyler       | 262,218 | 1.2771  | Taylor    | 168,977 | 0.8609  |
| 10   | Joseph      | 260,385 | 1.2681  | Megan     | 160,312 | 0.8168  |
| 11   | Brandon     | 259,299 | 1.2629  | Hannah    | 158,647 | 0.8083  |
| 12   | David       | 253,193 | 1.2332  | Kayla     | 155,844 | 0.7940  |

### Querying Baby Names

**Step 4** Choose between line, word, and character-based input.

The Social Security Administration data do not contain names with spaces, such as “Mary Jane”. Therefore, each data record contains exactly seven entries, as shown in the screen capture above. This input can be safely processed by reading words and numbers.

**Step 5** With line-oriented input, extract the required data.

We can skip this step because we don’t read a line at a time.

But suppose you decided in Step 4 to choose line-oriented input. Then you would need to break the input line into seven strings, converting five of them to numbers. This is quite tedious and it might well make you reconsider your choice.

**Step 6** Use methods to factor out common tasks.

In the pseudocode, we wrote **Repeat for girl part**. Clearly, there is a common task that calls for a helper method. It involves three tasks:

- Read the name, count, and percentage.
- Print the name if the total is less than 50 percent.
- Add the percentage to the total.

The last task poses a technical problem. In Java, it is not possible for a method to update a number parameter. Therefore, our method will receive a total and return the updated value. The updated value is then stored, like this:

```
boyTotal = processName(in, boyTotal, LIMIT);
girlTotal = processName(in, girlTotal, LIMIT);
```

As you can see, the method also needs to receive the Scanner object, so that it can read from it, and the percentage to be printed.

Here is the code of the method:

```
/*
 Reads name information, prints the name, and adjusts the total.
 @param in the input stream
 @param total the total percentage that should still be processed
 @param limit the cutoff for printing
 @return the adjusted total
*/
public static double processName(Scanner in, double total, double limit)
{
 String name = in.next();
 int count = in.nextInt();
 double percent = in.nextDouble();

 if (total < limit) { System.out.print(name + " "); }
 total = total + percent;
 return total;
}
```

The complete program is shown below.

Have a look at the program output. Remarkably, only 69 boy names and 153 girl names account for half of all births. That's good news for those who are in the business of producing personalized doodads. Exercise P7.1 asks you to study how this distribution changed over the years.

### **worked\_example\_1/BabyNames.java**

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5 /**
6 This program displays the most common baby names. Half of boys and girls in
7 the United States were given these names in the 1990s.
8 */
9 public class BabyNames
10 {
11 public static void main(String[] args) throws FileNotFoundException
12 {
13 final double LIMIT = 50;
14 Scanner in = new Scanner(new File("babynames.txt"));
15
16 double boyTotal = 0;
17 double girlTotal = 0;
18
19 while (boyTotal < LIMIT || girlTotal < LIMIT)
20 {
21 int rank = in.nextInt();
22 System.out.print(rank + " ");
23
24 boyTotal = processName(in, boyTotal, LIMIT);
25 girlTotal = processName(in, girlTotal, LIMIT);
26
27 System.out.println();
28 }
29 in.close();
30 }
31 }
```

```
32 /**
33 * Reads name information, prints the name, and adjusts the total.
34 * @param in the input stream
35 * @param total the total percentage that should still be processed
36 * @param limit the cutoff for printing
37 * @return the adjusted total
38 */
39 public static double processName(Scanner in, double total, double limit)
40 {
41 String name = in.next();
42 int count = in.nextInt();
43 double percent = in.nextDouble();
44
45 if (total < limit) { System.out.print(name + " "); }
46 total = total + percent;
47 return total;
48 }
49 }
```

### Program Run

```
1 Michael Jessica
2 Christopher Ashley
3 Matthew Emily
4 Joshua Sarah
5 Jacob Samantha
6 Nicholas Amanda
7 Andrew Brittany
8 Daniel Elizabeth
9 Tyler Taylor
10 Joseph Megan
...
68 Dustin Gabrielle
69 Noah Katie
70 Caitlin
71 Lindsey
...
150 Hayley
151 Rebekah
152 Jocelyn
153 Cassidy
```

# OBJECTS AND CLASSES

## CHAPTER GOALS

To understand the concepts of classes, objects, and encapsulation

To implement instance variables, methods, and constructors

To be able to design, implement, and test your own classes

To understand the behavior of object references, static variables, and static methods

## CHAPTER CONTENTS

### 8.1 OBJECT-ORIENTED PROGRAMMING 376

#### 8.2 IMPLEMENTING A SIMPLE CLASS 378

**SYN** Instance Variable Declaration 379

#### 8.3 SPECIFYING THE PUBLIC INTERFACE OF A CLASS 381

**ST1** The javadoc Utility 384

#### 8.4 DESIGNING THE DATA REPRESENTATION 385

#### 8.5 IMPLEMENTING INSTANCE METHODS 386

**SYN** Instance Methods 387

**PT1** All Instance Variables Should Be Private; Most Methods Should Be Public 388

#### 8.6 CONSTRUCTORS 389

**SYN** Constructors 390

**CE1** Trying to Call a Constructor 392

**CE2** Declaring a Constructor as void 393

**ST2** Overloading 393

#### 8.7 TESTING A CLASS 393

**HT1** Implementing a Class 395

**WE1** Implementing a Menu Class 

**VE1** Paying Off a Loan 



© Cameron Strathdee/iStockphoto.

#### 8.8 PROBLEM SOLVING: TRACING OBJECTS 399

**C&S** Open Source and Free Software 402

#### 8.9 OBJECT REFERENCES 403

**CE3** Forgetting to Initialize Object References in a Constructor 407

**ST3** Calling One Constructor from Another 408

#### 8.10 STATIC VARIABLES AND METHODS 408

#### 8.11 PROBLEM SOLVING: PATTERNS FOR OBJECT DATA 410

**C&S** Electronic Voting Machines 416

**VE2** Modeling a Robot Escaping from a Maze 

#### 8.12 PACKAGES 417

**SYN** Package Specification 419

**CE4** Confusing Dots 420

**ST4** Package Access 421

**HT2** Programming with Packages 421



© Cameron Strathdee/iStockphoto.

This chapter introduces you to object-oriented programming, an important technique for writing complex programs. In an object-oriented program, you don't simply manipulate numbers and strings, but you work with objects that are meaningful for your application. Objects with the same behavior (such as the windmills shown here) are grouped into classes. A programmer provides the desired behavior by specifying and implementing methods for these classes. In this chapter, you will learn how to discover, specify, and implement your own classes, and how to use them in your programs.

## 8.1 Object-Oriented Programming

You have learned how to structure your programs by decomposing tasks into methods. This is an excellent practice, but experience shows that it does not go far enough. It is difficult to understand and update a program that consists of a large collection of methods.

To overcome this problem, computer scientists invented **object-oriented programming**, a programming style in which tasks are solved by collaborating objects. Each object has its own set of data, together with a set of methods that act upon the data.

You have already experienced this programming style when you used strings, the `System.out` object, or a `Scanner` object. Each of these objects has a set of methods. For example, you can use the `length` and `substring` methods to work with `String` objects.

When you develop an object-oriented program, you create your own objects that describe what is important in your application. For example, in a student database you might work with `Student` and `Course` objects. Of course, then you must supply methods for these objects.

In Java, a programmer doesn't implement a single object. Instead, the programmer provides a **class**. A class describes a set of objects with the same behavior. For example, the `String` class describes the behavior of all strings. The class specifies how

A class describes a set of objects with the same behavior.



Media Bakery.

a string stores its characters, which methods can be used with strings, and how the methods are implemented.

In contrast, the `PrintStream` class describes the behavior of objects that can be used to produce output. One such object is `System.out`, and you have seen in Chapter 7 how to create `PrintStream` objects that send output to a file.

Each class defines a specific set of methods that you can use with its objects. For example, when you have a `String` object, you can invoke the `length` method:

```
"Hello, World".length()
```

We say that the `length` method is a method of the `String` class. The `PrintStream` class has a different set of methods. For example, the call

```
System.out.length()
```

would be illegal—the `PrintStream` class has no `length` method. However, `PrintStream` has a `println` method, and the call

```
out.println("Hello, World!")
```

is legal.

The set of all methods provided by a class, together with a description of their behavior, is called the **public interface** of the class.

When you work with an object of a class, you do not know how the object stores its data, or how the methods are implemented. You need not know how a `String` organizes a character sequence, or how a `Writer` object sends data to a file. All you need to know is the public interface—which methods you can apply, and what these methods do. The process of providing a public interface, while hiding the implementation details, is called **encapsulation**.

When you design your own classes, you will use encapsulation. That is, you will specify a set of public methods and hide the implementation details. Other programmers on your team can then use your classes without having to know their implementations, just as you are able to make use of the `String` and `PrintStream` classes.

If you work on a program that is being developed over a long period of time, it is common for implementation details to change, usually to make objects more efficient or more capable. When the implementation is hidden, the improvements do not affect the programmers that use the objects.

Every class has a public interface: a collection of methods through which the objects of the class can be manipulated.

Encapsulation is the act of providing a public interface and hiding the implementation details.

Encapsulation enables changes in the implementation without affecting users of a class.

*You can drive a car by operating the steering wheel and pedals, without knowing how the engine works. Similarly, you use an object through its methods. The implementation is hidden.*



© Damir Cudic/Stockphoto.

*A driver of an electric car doesn't have to learn new controls even though the car engine is very different. Neither does the programmer who uses an object with an improved implementation—as long as the same methods are used.*



© iStockphoto.com/Christian Waadt.



1. Is the method call "Hello, World".println() legal? Why or why not?
2. When using a String object, you do not know how it stores its characters. How can you access them?
3. Describe a way in which a String object might store its characters.
4. Suppose the providers of your Java compiler decide to change the way that a String object stores its characters, and they update the String method implementations accordingly. Which parts of your code do you need to change when you get the new compiler?

**Practice It** Now you can try these exercises at the end of the chapter: R8.1, R8.4.

## 8.2 Implementing a Simple Class

In this section, we look at the implementation of a very simple class. You will see how objects store their data, and how methods access the data of an object. Knowing how a very simple class operates will help you design and implement more complex classes later in this chapter.

Our first example is a class that models a *tally counter*, a mechanical device that is used to count people—for example, to find out how many people attend a concert or board a bus (see Figure 1).

Whenever the operator pushes a button, the counter value advances by one. We model this operation with a count method. A physical counter has a display to show the current value. In our simulation, we use a getValue method instead.



© Jasmin Awad/iStockphoto.

**Figure 1** A Tally Counter

Here is an example of using the Counter class. As you know from working with classes such as Scanner and ArrayList, you use the new operator to construct new objects of a class. To construct an object of the Counter class, you call

```
Counter tally = new Counter();
```

Now tally refers to a Counter object whose value is zero.

Next, we invoke methods on our object. First, we invoke the count method twice, simulating two button pushes. Then we invoke the getValue method to check how many times the button was pushed.

```
tally.count();
tally.count();
int result = tally.getValue(); // Sets result to 2
```

We can invoke the methods again, and the result will be different.

```
tally.count();
tally.count();
result = tally.getValue(); // Sets result to 4
```

As you can see, the tally object remembers the effect of prior method calls.

By specifying the methods of the Counter class, we have now specified the **behavior** of Counter objects. In order to produce this behavior, each object needs internal data, called the **state** of the object. In this simple example, the object state is very simple. It is the value that keeps track of how many times the counter has been advanced.

An object stores its state in **instance variables**. An *instance* of a class is an object of the class. Thus, an instance variable is a storage location that is present in each object of the class.

You specify instance variables in the class declaration:

```
public class Counter
{
 private int value;
 ...
}
```

An instance variable declaration consists of the following parts:

- A **modifier** (private)
- The **type** of the instance variable (such as int)
- The name of the instance variable (such as value)

## Syntax 8.1 Instance Variable Declaration

*Syntax*

```
public class ClassName
{
 private typeName variableName;
 ...
}
```

Instance variables should always be private.

```
public class Counter
{
 private int value;
 ...
}
```

Each object of this class has a separate copy of this instance variable.

Type of the variable

Each object of a class has its own set of instance variables.

An instance method can access the instance variables of the object on which it acts.

A private instance variable can only be accessed by the methods of its own class.

Each object of a class has its own set of instance variables. For example, if concertCounter and boardingCounter are two objects of the Counter class, then each object has its own value variable (see Figure 2).

As you will see in Section 8.6, the instance variable value is set to 0 when a Counter object is constructed.

Next, let us have a quick look at the implementation of the methods of the Counter class. The count method advances the counter value by 1.

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

We will cover the syntax of the method header in Section 8.3. For now, focus on the body of the method inside the braces.

Note how the count method increments the instance variable value. *Which* instance variable? The one belonging to the object on which the method is invoked. For example, consider the call

```
concertCounter.count();
```

This call advances the value variable of the concertCounter object.

The methods that you invoke on an object are called **instance methods** to distinguish them from the static methods of Chapter 5.

Finally, look at the other instance method of the Counter class. The getValue method returns the current value:

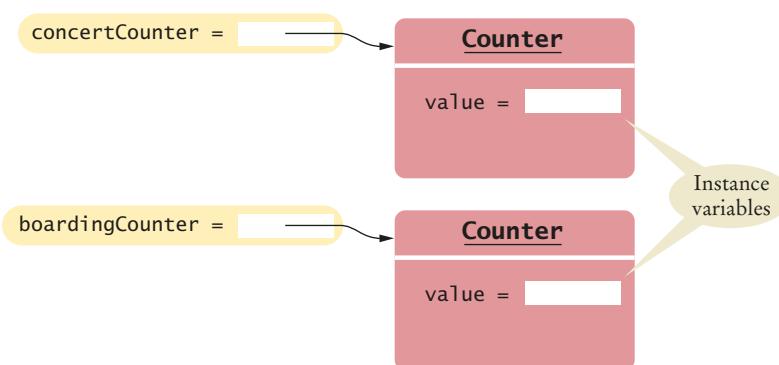
```
public int getValue()
{
 return value;
}
```

This method is required so that users of the Counter class can find out how often a particular counter has been clicked. A user cannot simply access the value instance variable. That variable has been declared with the access specifier **private**.

The private specifier restricts access to the methods of the *same class*. For example, the value variable can be accessed by the count and getValue methods of the Counter class but not by a method of another class. Those other methods need to use the getValue method if they want to find out the counter's value, or the count method if they want to change it.

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download the complete Counter class and a CounterDemo program.



**Figure 2** Instance Variables



© Mark Evans/Stockphoto.

*These clocks have common behavior, but each of them has a different state. Similarly, objects of a class can have their instance variables set to different values.*

Private instance variables are an essential part of encapsulation. They allow a programmer to hide the implementation of a class from a class user.



#### SELF CHECK

5. Supply the body of a method `public void reset()` that resets the counter back to zero.
6. Consider a change to the implementation of the counter. Instead of using an integer counter, we use a string of | characters to keep track of the clicks, just like a human might do.

```
public class Counter
{
 private String strokes = "";
 public void count()
 {
 strokes = strokes + "|";
 }
 ...
}
```

How do you implement the `getValue` method with this data representation?

7. Suppose another programmer has used the original `Counter` class. What changes does that programmer have to make in order to use the modified class?
8. Suppose you use a class `Clock` with private instance variables `hours` and `minutes`. How can you access these variables in your program?

**Practice It** Now you can try these exercises at the end of the chapter: E8.1, E8.2.

## 8.3 Specifying the Public Interface of a Class

When designing a class, you start by specifying its **public interface**. The public interface of a class consists of all methods that a user of the class may want to apply to its objects.

Let's consider a simple example. We want to use objects that simulate cash registers. A cashier who rings up a sale presses a key to start the sale, then rings up each item. A display shows the amount owed as well as the total number of items purchased.

You can use method headers and method comments to specify the public interface of a class.

In our simulation, we want to call the following methods on a cash register object:

- Add the price of an item.
- Get the total amount owed, and the count of items purchased.
- Clear the cash register to start a new sale.

Here is an outline of the `CashRegister` class. We supply comments for all of the methods to document their purpose.

```

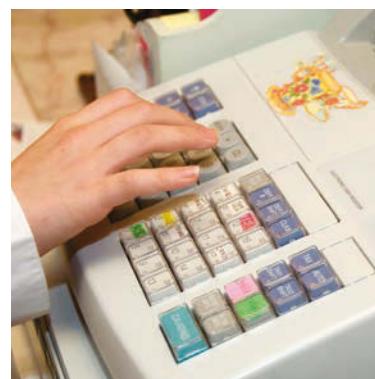
 /**
 * A simulated cash register that tracks the item
 * count and the total amount due.
 */
public class CashRegister
{
 private data—see Section 8.4

 /**
 * Adds an item to this cash register.
 * @param price the price of this item
 */
 public void addItem(double price)
 {
 implementation—see Section 8.5
 }

 /**
 * Gets the price of all items in the current sale.
 * @return the total price
 */
 public double getTotal()
 {
 implementation—see Section 8.5
 }

 /**
 * Gets the number of items in the current sale.
 * @return the item count
 */
 public int getCount()
 {
 implementation—see Section 8.5
 }

 /**
 * Clears the item count and the total.
 */
 public void clear()
 {
 implementation—see Section 8.5
 }
}
```



© James Richey/iStockphoto.

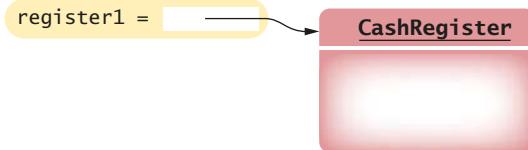
#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download the documentation of the public interface of the `CashRegister` class.

The method declarations and comments make up the *public interface* of the class. The data and the method bodies make up the *private implementation* of the class.

Note that the methods of the `CashRegister` class are instance methods. They are *not* declared as `static`. You invoke them on objects (or instances) of the `CashRegister` class.

**Figure 3**  
An Object Reference  
and an Object



To see an instance method in action, we first need to construct an object:

```
CashRegister register1 = new CashRegister();
// Constructs a CashRegister object
```

This statement initializes the `register1` variable with a reference to a new `CashRegister` object—see Figure 3. (We discuss the process of object construction in Section 8.6 and object references in Section 8.9.)

Once the object has been constructed, we are ready to invoke a method:

```
register1.addItem(1.95); // Invokes a method
```

A mutator method  
changes the object  
on which it operates.

An accessor method  
does not change  
the object on which  
it operates.

When you look at the public interface of a class, it is useful to classify its methods as *mutators* and *accessors*. A **mutator** method modifies the object on which it operates. The `CashRegister` class has two mutators: `addItem` and `clear`. After you call either of these methods, the object has changed. You can observe that change by calling the `getTotal` or `getCount` method.

An **accessor** method queries the object for some information without changing it. The `CashRegister` class has two accessors: `getTotal` and `getCount`. Applying either of these methods to a `CashRegister` object simply returns a value and does not modify the object. For example, the following statement prints the current total and count:

```
System.out.println(register1.getTotal() + " " + register1.getCount());
```

Now we know *what* a `CashRegister` object can do, but not *how* it does it. Of course, to use `CashRegister` objects in our programs, we don't need to know.

In the next sections, you will see how the `CashRegister` class is implemented.

### SELF CHECK



9. What does the following code segment print?

```
CashRegister reg = new CashRegister();
reg.clear();
reg.addItem(0.95);
reg.addItem(0.95);
System.out.println(reg.getCount() + " " + reg.getTotal());
```

10. What is wrong with the following code segment?

```
CashRegister reg = new CashRegister();
reg.clear();
reg.addItem(0.95);
System.out.println(reg.getAmountDue());
```

11. Declare a method `getDollars` of the `CashRegister` class that yields the amount of the total sale as a dollar value without the cents.

12. Name two accessor methods of the `String` class.

13. Is the `nextInt` method of the `Scanner` class an accessor or a mutator?

14. Provide documentation comments for the `Counter` class of Section 8.2.

### Practice It

Now you can try these exercises at the end of the chapter: R8.2, R8.8.

## Special Topic 8.1

**The javadoc Utility**

The javadoc utility formats documentation comments into a neat set of documents that you can view in a web browser. It makes good use of the seemingly repetitive phrases. The first sentence of each method comment is used for a *summary table* of all methods of your class (see Figure 4). The @param and @return comments are neatly formatted in the detail description of each method (see Figure 5). If you omit any of the comments, then javadoc generates documents that look strangely empty.

This documentation format may look familiar. It is the same format that is used in the official Java documentation. The programmers who implement the Java library use javadoc themselves. They too document every class, every method, every parameter, and every return value, and then use javadoc to extract the documentation.

Many integrated programming environments can execute javadoc for you. Alternatively, you can invoke the javadoc utility from a shell window, by issuing the command

```
javadoc MyClass.java
```

| Method Summary |                                                                                |
|----------------|--------------------------------------------------------------------------------|
| void           | <a href="#">addItem(double price)</a><br>Adds an item to this cash register.   |
| void           | <a href="#">clear()</a><br>Clears the item count and the total.                |
| int            | <a href="#">getCount()</a><br>Gets the number of items in the current sale.    |
| double         | <a href="#">getTotal()</a><br>Gets the price of all items in the current sale. |

**Figure 4** A Method Summary Generated by javadoc

**addItem**

```
public void addItem(double price)
```

Adds an item to this cash register.

**Parameters:**

price - the price of this item

---

**getTotal**

**Figure 5** Method Detail Generated by javadoc

The javadoc utility produces files such as `MyClass.html` in HTML format, which you can inspect in a browser. You can use hyperlinks to navigate to other classes and methods.

You can run javadoc before implementing any methods. Just leave all the method bodies empty. Don't run the compiler—it would complain about missing return values. Simply run javadoc on your file to generate the documentation for the public interface that you are about to implement.

The javadoc tool is wonderful because it does one thing right: It allows you to put the documentation *together with your code*. That way, when you update your programs, you can see right away which documentation needs to be updated. Hopefully, you will update it right then and there. Afterward, run javadoc again and get updated information that is timely and nicely formatted.

## 8.4 Designing the Data Representation

An object stores its data in **instance variables**. These are variables that are declared inside the class (see Syntax 8.1).

When implementing a class, you have to determine which data each object needs to store. The object needs to have all the information necessary to carry out any method call.

Go through all methods and consider their data requirements. It is a good idea to start with the accessor methods. For example, a `CashRegister` object must be able to return the correct value for the `getTotal` method. That means, it must either store all entered prices and compute the total in the method call, or it must store the total.

Now apply the same reasoning to the `getCount` method. If the cash register stores all entered prices, it can count them in the `getCount` method. Otherwise, you need to have a variable for the count.

The `addItem` method receives a price as an argument, and it must record the price. If the `CashRegister` object stores an array of entered prices, then the `addItem` method appends the price. On the other hand, if we decide to store just the item total and count, then the `addItem` method updates these two variables.

Finally, the `clear` method must prepare the cash register for the next sale, either by emptying the array of prices or by setting the total and count to zero.

We have now discovered two different ways of representing the data that the object needs. Either of them will work, and we have to make a choice. We will choose the simpler one: variables for the total price and the item count. (Other options are explored in Exercises P8.10 and P8.11.)

```
int itemCount;
double totalPrice;
```

The instance variables are declared in the class, but outside any methods, with the `private` modifier:

```
public class CashRegister
{
```

Each accessor method must either retrieve a stored value or compute the result.

Commonly, there is more than one way of representing the data of an object, and you must make a choice.



© iStockphoto.com/mgini.

*Like a wilderness explorer who needs to carry all items that may be needed, an object needs to store the data required for any method calls.*

Be sure that your data representation supports method calls in any order.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to get the CashRegister class with instance variables.

**SELF CHECK**

```
private int itemCount;
private double totalPrice;
. . .
}
```

Note that method calls can come in *any order*. For example, consider the CashRegister class. After calling

```
register1.getTotal()
```

a program can make another call to

```
register1.addItem(1.95)
```

You should not assume that you can clear the sum in a call to getTotal. Your data representation should allow for method calls that come in arbitrary order, in the same way that occupants of a car can push the various buttons and levers in any order they choose.

- 15.** What is wrong with this code segment?

```
CashRegister register2 = new CashRegister();
register2.clear();
register2.addItem(0.95);
System.out.println(register2.totalPrice);
```

- 16.** Consider a class Time that represents a point in time, such as 9 A.M. or 3:30 P.M. Give two sets of instance variables that can be used for implementing the Time class. (*Hint for the second set:* Military time.)
- 17.** Suppose the implementor of the Time class changes from one implementation strategy to another, keeping the public interface unchanged. What do the programmers who use the Time class need to do?
- 18.** Consider a class Grade that represents a letter grade, such as A+ or B. Give two different sets of instance variables that can be used for implementing the Grade class.

**Practice It** Now you can try these exercises at the end of the chapter: R8.6, R8.16.

## 8.5 Implementing Instance Methods

In Chapter 5, you learned how to implement static methods. A static method carries out some work, but it is not invoked on an object. For example, to call the cubeVolume method from Section 5.2, you make a call such as

```
double result = cubeVolume(2);
```

In contrast, an instance method is invoked on an object. For example, here we call the addItem method on a CashRegister object:

```
register1.addItem(1.95);
```

In this call, the addItem method updates the instance variables of the register1 object.

Here is the implementation of the addItem method of the CashRegister class. (You can find the remaining methods at the end of the next section.)

```
public void addItem(double price)
{
 itemCount++;
 totalPrice = totalPrice + price;
}
```

## Syntax 8.2 Instance Methods

```
Syntax modifiers returnType methodName(parameterType parameterName, . . .)
{
 method body
}
```

```
public class CashRegister
{
 .
 .
 public void addItem(double price)
 {
 itemCount++;
 totalPrice = totalPrice + price;
 }
 .
 .
}
```

The object on which an instance method is applied is the implicit parameter.

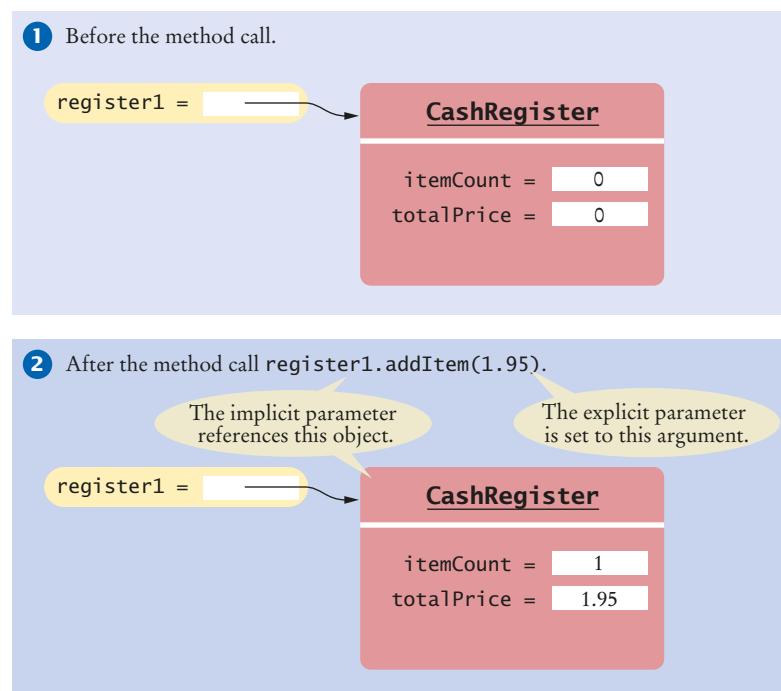
Whenever you use an instance variable, such as `itemCount` or `totalPrice`, in a method, it denotes that instance variable of *the object on which an instance method is invoked*. For example, consider the call

```
register1.addItem(1.95);
```

The first statement in the `addItem` method is

```
itemCount++;
```

Which `itemCount` is incremented? In this call, it is the `itemCount` of the `register1` object. (See Figure 6.)



**Figure 6**  
Implicit and Explicit Parameters

*When an item is added, it affects the instance variables of the cash register object on which the method is invoked.*



© Glow Images/Getty Images, Inc.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download the CashRegister class with method implementations.

Explicit parameters of a method are listed in the method declaration.

**SELF CHECK**

19. What are the values of register1.itemCount, register1.totalPrice, register2.itemCount, and register2.totalPrice after these statements?  

```
CashRegister register1 = new CashRegister();
register1.addItem(0.90);
register1.addItem(0.95);
CashRegister register2 = new CashRegister();
register2.addItem(1.90);
```
20. Implement a method `getDollars` of the `CashRegister` class that yields the amount of the total sale as a dollar value without the cents.
21. Consider the `substring` method of the `String` class that is described in Section 2.5.6. How many parameters does it have, and what are their types?
22. Consider the `length` method of the `String` class. How many parameters does it have, and what are their types?

**Practice It** Now you can try these exercises at the end of the chapter: R8.10, P8.10, P8.11, P8.12.

**Programming Tip 8.1****All Instance Variables Should Be Private; Most Methods Should Be Public**

It is possible to declare instance variables as public, but you should not do that in your own code. Always use encapsulation, with private instance variables that are manipulated with methods.

Typically, methods are public. However, sometimes you have a method that is used only as a helper method by other methods. In that case, you can make the helper method private. Simply use the `private` reserved word when declaring the method.

## 8.6 Constructors

A constructor initializes the instance variables of an object.

A constructor is invoked when an object is created with the new operator.

The name of a constructor is the same as the class name.

A class can have multiple constructors.

A **constructor** initializes the instance variables of an object. The constructor is automatically called whenever an object is created with the new operator.

You have seen the new operator in Chapter 2. It is used whenever a new object is required. For example, the expression new Scanner(System.in) in the statement

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

constructs a new object of the Scanner class. Specifically, a constructor of the Scanner class is called with the argument System.in. That constructor initializes the Scanner object.

The name of a constructor is identical to the name of its class. For example:

```
public class CashRegister
{
 ...
 /**
 * Constructs a cash register with cleared item count and total.
 */
 public CashRegister() // A constructor
 {
 itemCount = 0;
 totalPrice = 0;
 }
}
```

Constructors never return values, but you do not use the void reserved word when declaring them.

Many classes have more than one constructor. This allows you to declare objects in different ways. In How To 8.1, we will develop a BankAccount class that has two constructors:

```
public class BankAccount
{
 ...
 /**
 * Constructs a bank account with a zero balance.
 */
 public BankAccount() { . . . }

 /**
 * Constructs a bank account with a given balance.
 * @param initialBalance the initial balance
 */
 public BankAccount(double initialBalance) { . . . }
}
```

Both constructors have the same name as the class, BankAccount. The first constructor has no parameter variables, whereas the second constructor has a parameter variable of type double.

When you construct an object, the compiler chooses the constructor that matches the arguments that you supply. For example,

```
BankAccount joesAccount = new BankAccount();
// Uses BankAccount() constructor
BankAccount lisasAccount = new BankAccount(499.95);
// Uses BankAccount(double) constructor
```

The compiler picks the constructor that matches the construction arguments.

## Syntax 8.3 Constructors

```

public class BankAccount
{
 private double balance;

 public BankAccount()
 {
 balance = 0;
 }

 public BankAccount(double initialBalance)
 {
 balance = initialBalance;
 }

 ...
}

```

**A constructor has no return type, not even void.**

**These constructors initialize the balance instance variable.**

**A constructor has the same name as the class.**

**This constructor is picked for the expression new BankAccount(499.95).**

By default, numbers are initialized as 0, Booleans as false, and object references as null.

If you do not initialize an instance variable in a constructor, it is automatically set to a default value:

- Numbers are set to zero.
- Boolean variables are initialized as false.
- Object and array references are set to the special value `null` that indicates that no object is associated with the variable (see Section 8.9). This is usually not desirable, and you should initialize object references in your constructors (see Common Error 8.3 on page 407).

In this regard, instance variables differ from local variables declared inside methods. The computer reports an error if you use a local variable that has not been explicitly initialized.

If you do not supply any constructor for a class, the compiler automatically generates a constructor. That constructor has no arguments, and it initializes all instance variables with their default values. Therefore, every class has at least one constructor.

You have now encountered all concepts that are necessary to implement the `CashRegister` class.

*A constructor is like a set of assembly instructions for an object.*



© Ann Marie Kurtz/Stockphoto.

The complete code for the class is given here. In the next section, you will see how to test the class.

### sec06/CashRegister.java

```
1 /**
2 * A simulated cash register that tracks the item count and
3 * the total amount due.
4 */
5 public class CashRegister
6 {
7 private int itemCount;
8 private double totalPrice;
9
10 /**
11 Constructs a cash register with cleared item count and total.
12 */
13 public CashRegister()
14 {
15 itemCount = 0;
16 totalPrice = 0;
17 }
18
19 /**
20 Adds an item to this cash register.
21 @param price the price of this item
22 */
23 public void addItem(double price)
24 {
25 itemCount++;
26 totalPrice = totalPrice + price;
27 }
28
29 /**
30 Gets the price of all items in the current sale.
31 @return the total amount
32 */
33 public double getTotal()
34 {
35 return totalPrice;
36 }
37
38 /**
39 Gets the number of items in the current sale.
40 @return the item count
41 */
42 public int getCount()
43 {
44 return itemCount;
45 }
46
47 /**
48 Clears the item count and the total.
49 */
50 public void clear()
51 {
52 itemCount = 0;
53 totalPrice = 0;
54 }
55 }
```



- 23.** Consider this class:

```
public class Person
{
 private String name;

 public Person(String firstName, String lastName)
 {
 name = lastName + ", " + firstName;
 }
 . .
}
```

If an object is constructed as

```
Person harry = new Person("Harry", "Morgan");
```

what is its name instance variable?

- 24.** Provide an implementation for a Person constructor so that after the call

```
Person p = new Person();
```

the name instance variable of p is "unknown".

- 25.** What happens if you supply no constructor for the CashRegister class?

- 26.** Consider the following class:

```
public class Item
{
 private String description;
 private double price;

 public Item() { . . . }
 // Additional methods omitted
}
```

Provide an implementation for the constructor.

- 27.** Which constructors should be supplied in the Item class so that each of the following declarations compiles?

- a.** Item item2 = new Item("Corn flakes");
- b.** Item item3 = new Item(3.95);
- c.** Item item4 = new Item("Corn flakes", 3.95);
- d.** Item item1 = new Item();
- e.** Item item5;

**Practice It** Now you can try these exercises at the end of the chapter: R8.12, E8.6, P8.5.

### Common Error 8.1



#### Trying to Call a Constructor

A constructor is not a method. You must use it in combination with the new reserved word:

```
CashRegister register1 = new CashRegister();
```

After an object has been constructed, you cannot invoke the constructor on that object again. For example, you cannot call the constructor to clear an object:

```
. .
register1.CashRegister(); // Error
```

It is true that the constructor can set a *new* CashRegister object to the cleared state, but you cannot invoke a constructor on an *existing* object. However, you can replace the object with a new one:

```
register1 = new CashRegister(); // OK
```

### Common Error 8.2



#### Declaring a Constructor as void

Do not use the `void` reserved word when you declare a constructor:

```
public void BankAccount() // Error—don't use void!
```

This would declare a method with return type `void` and *not* a constructor. Unfortunately, the Java compiler does not consider this a syntax error.

### Special Topic 8.2



#### Overloading

When the same method name is used for more than one method, then the name is **overloaded**. In Java you can overload method names provided that the parameter types are different. For example, you can declare two methods, both called `print`:

```
public void print(CashRegister register)
public void print(BankAccount account)
```

When the `print` method is called,

```
print(x);
```

the compiler looks at the type of `x`. If `x` is a `CashRegister` object, the first method is called. If `x` is an `BankAccount` object, the second method is called. If `x` is neither, the compiler generates an error.

We have not used the overloading feature in this book. Instead, we gave each method a unique name, such as `printRegister` or `printAccount`. However, we have no choice with constructors. Java demands that the name of a constructor equal the name of the class. If a class has more than one constructor, then that name must be overloaded.

## 8.7 Testing a Class

In the preceding section, we completed the implementation of the `CashRegister` class. What can you do with it? Of course, you can compile the file `CashRegister.java`. However, you can't *execute* the `CashRegister` class. It doesn't contain a `main` method. That is normal—most classes don't contain a `main` method. They are meant to be combined with a class that has a `main` method.

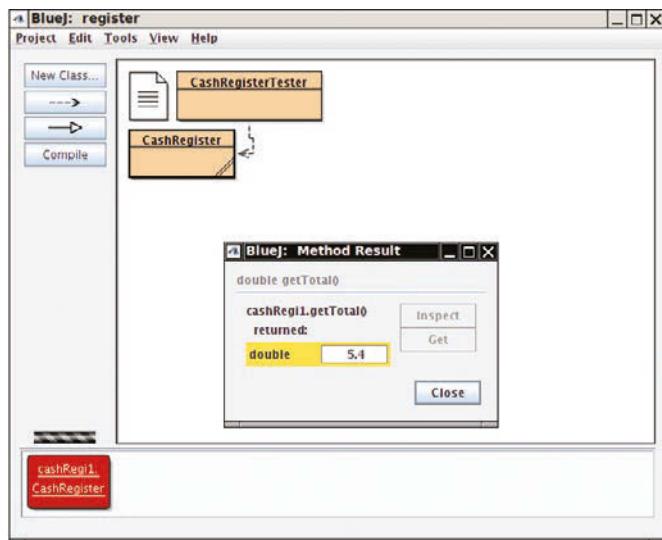
In the long run, your class may become a part of a larger program that interacts with users, stores data in files, and so on. However, before integrating a class into a program, it is always a good idea to test it in isolation. Testing in isolation, outside a complete program, is called **unit testing**.

A unit test verifies that a class works correctly in isolation, outside a complete program.



© Chris Ferting/Stockphoto.

An engineer tests a part in isolation. This is an example of unit testing.



**Figure 7** The Return Value of the `getTotal` Method in BlueJ

To test your class, you have two choices. Some interactive development environments, such as BlueJ (<http://bluej.org>) and Dr. Java (<http://drjava.org>), have commands for constructing objects and invoking methods. Then you can test a class simply by constructing an object, calling methods, and verifying that you get the expected return values. Figure 7 shows the result of calling the `getTotal` method on a `CashRegister` object in BlueJ.

Alternatively, you can write a *tester class*. A tester class is a class with a `main` method that contains statements to run methods of another class. A tester class typically carries out the following steps:

1. Construct one or more objects of the class that is being tested.
2. Invoke one or more methods.
3. Print out one or more results.
4. Print the expected results.

Here is a class to run methods of the `CashRegister` class. The `main` method constructs an object of type `CashRegister`, invokes the `addItem` method three times, and then displays the result of the `getCount` and `getTotal` methods.

### sec07/CashRegisterTester.java

```

1 /**
2 * This program tests the CashRegister class.
3 */
4 public class CashRegisterTester
5 {
6 public static void main(String[] args)
7 {
8 CashRegister register1 = new CashRegister();
9 register1.addItem(1.95);
10 register1.addItem(0.95);
11 register1.addItem(2.50);
12 System.out.println(register1.getCount());
13 System.out.println("Expected: 3");

```

To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.

```

14 System.out.printf("%.2f%n", register1.getTotal());
15 System.out.println("Expected: 5.40");
16 }
17 }
```

### Program Run

```

3
Expected: 3
5.40
Expected: 5.40
```

Determining the expected result in advance is an important part of testing.

In our sample program, we add three items totaling \$5.40. When displaying the method results, we also display messages that describe the values we expect to see.

This is a very important step. You want to spend some time thinking about what the expected result is before you run a test program. This thought process will help you understand how your program should behave, and it can help you track down errors at an early stage.

To produce a program, you need to combine the `CashRegister` and `CashRegisterTester` classes. The details for building the program depend on your compiler and development environment. In most environments, you need to carry out these steps:

1. Make a new subfolder for your program.
2. Make two files, one for each class.
3. Compile both files.
4. Run the test program.

Many students are surprised that such a simple program contains two classes. However, this is normal. The two classes have entirely different purposes. The `CashRegister` class describes objects that model cash registers. The `CashRegisterTester` class runs a test that puts a `CashRegister` object through its paces.

- SELF CHECK**
- 
28. How would you enhance the tester class to test the `clear` method?
  29. When you run the `CashRegisterTester` program, how many objects of class `CashRegister` are constructed? How many objects of type `CashRegisterTester`?
  30. Why is the `CashRegisterTester` class unnecessary in development environments that allow interactive testing, such as BlueJ?

**Practice It** Now you can try these exercises at the end of the chapter: P8.8, P8.9, P8.15.

### HOW TO 8.1



### Implementing a Class

A very common task is to implement a class whose objects can carry out a set of specified actions. This How To walks you through the necessary steps.

**Problem Statement** Consider a class that simulates a bank account. Customers can deposit and withdraw funds. If sufficient funds are not available for withdrawal, a \$10 overdraft penalty is charged. At the end of the month, interest is added to the account. The interest rate can vary every month.



© Mark Evans/iStockphoto.

**Step 1** Get an informal list of the responsibilities of your objects.

The following responsibilities are mentioned in the problem statement:

- Deposit funds.**
- Withdraw funds.**
- Add interest.**

Now look for hidden responsibilities that aren't part of the problem description. Which mundane activities need to happen, such as clearing the cash register at the beginning of each sale?

In the case of the bank account, we need to be able to find out how much money is in a given account. Let's note that responsibility:

**Get balance.**

Be careful that you restrict yourself to features that are actually required in the problem. With real-world items, such as cash registers or bank accounts, there are potentially dozens of features that might be worth implementing. But your job is not to faithfully model the real world. You need to determine only those responsibilities that you need for solving your specific problem.

Real bank accounts have account numbers and one or more owners. In this problem, we are not asked to implement those aspects.

**Step 2** Specify the public interface.

Turn the list in Step 1 into a set of methods, with specific types for the parameter variables and the return values. Many programmers find this step simpler if they write out method calls that are applied to a sample object, like this:

```
BankAccount harrysAccount = new BankAccount();
harrysAccount.deposit(500);
harrysAccount.withdraw(2000); // Should fail and charge penalty
harrysAccount.addInterest(1); // 1% interest
double balance = harrysAccount.getBalance();
```

We need to supply parameter variables and determine which methods are accessors and mutators.

To deposit or withdraw money, one needs to know the amount of the deposit or withdrawal.

```
void deposit(double amount)
void withdraw(double amount)
```

To add interest, one needs to know the interest rate that is to be applied:

```
void addInterest(double rate)
```

Clearly, all these methods are mutators because they change the balance.

Finally, we have

```
double getBalance()
```

This method is an accessor because inquiring about the balance does not change it.

To complete the public interface, you need to specify the constructors. Ask yourself what information you need in order to construct an object of your class. Sometimes you will want two constructors: one that sets all instance variables to a default and one that sets them to user-supplied values.

In our case, the constructor with no arguments makes an account with a zero balance. Let us also supply a constructor with an initial balance.

Here is the complete public interface:

- Constructors

```
public BankAccount()
public BankAccount(double initialBalance)
```

- Mutators

```
public void deposit(double amount)
public void withdraw(double amount)
public void addInterest(double rate)
```

- Accessors

```
public double getBalance()
```

**Step 3** Document the public interface.

Supply a documentation comment for the class, then comment each method.

```
/**
 * A bank account has a balance that can be changed by
 * deposits and withdrawals.
 */
public class BankAccount
{
 /**
 * Constructs a bank account with a zero balance.
 */
 public BankAccount()

 /**
 * Constructs a bank account with a given balance.
 * @param initialBalance the initial balance
 */
 public BankAccount(double initialBalance)

 /**
 * Deposits money into this account.
 * @param amount the amount to deposit
 */
 public void deposit(double amount)

 /**
 * Makes a withdrawal from this account, or charges a penalty if
 * sufficient funds are not available.
 * @param amount the amount of the withdrawal
 */
 public void withdraw(double amount)

 /**
 * Adds interest to this account.
 * @param rate the interest rate in percent
 */
 public void addInterest(double rate)

 /**
 * Gets the current balance of this account.
 * @return the current balance
 */
 public double getBalance()
}
```

**Step 4** Determine instance variables.

Ask yourself what information an object needs to store to do its job. The object needs to be able to process every method using just its instance variables and the method arguments. Go through each method, perhaps starting with a simple one or an interesting one, and ask yourself what the object needs to carry out the method's task. Which data items are required in addition to the method arguments? Make instance variables for those data items.

Clearly we need to store the bank balance.

```
public class BankAccount
{
 private double balance;
 ...
}
```

Do we need to store the interest rate? No—it varies every month, and is supplied as an argument to `addInterest`. What about the withdrawal penalty? The problem description states that it is a fixed \$10, so we need not store it. If the penalty could vary over time, as is the case with most real bank accounts, we would need to store it somewhere (perhaps in a `Bank` object), but it is not our job to model every aspect of the real world.

### Step 5 Implement constructors and methods.

In this step, you implement the constructors and methods in your class, one at a time.

Let's start with a simple one:

```
public double getBalance()
{
 return balance;
}
```

The `deposit` method is a bit more interesting:

```
public void deposit(double amount)
{
 balance = balance + amount;
}
```

The `withdraw` method needs to charge a penalty if sufficient funds are not available:

```
public void withdraw(double amount)
{
 final double PENALTY = 10;
 if (amount > balance)
 {
 balance = balance - PENALTY;
 }
 else
 {
 balance = balance - amount;
 }
}
```

Finally, here is the `addInterest` method. We compute the interest and then add it to the balance:

```
public void addInterest(double rate)
{
 double amount = balance * rate / 100;
 balance = balance + amount;
}
```

The constructors are once again quite simple:

```
public BankAccount()
{
 balance = 0;
}

public BankAccount(double initialBalance)
{
 balance = initialBalance;
}
```

If you find that you have trouble with the implementation of some of your methods, you may need to rethink your choice of instance variables. It is common for a beginner to start out with a set of instance variables that cannot accurately describe the state of an object. Don't hesitate to go back and rethink your implementation strategy.

Once you have completed the implementation, compile your class and fix any compiler errors.

### Step 6

Test your class.

Write a short tester program and execute it. The tester program should carry out the method calls that you found in Step 2.

Here is a simple test program that exercises all methods:

```
public class BankAccountTester
{
 public static void main(String[] args)
 {
 BankAccount harrysAccount = new BankAccount(1000);
 harrysAccount.deposit(500); // Balance is now $1500
 harrysAccount.withdraw(2000); // Balance is now $1490
 harrysAccount.addInterest(1); // Balance is now $1490 + 14.90
 System.out.printf("%.2f%n", harrysAccount.getBalance());
 System.out.println("Expected: 1504.90");
 }
}
```

### Program Run

```
1504.90
Expected: 1504.90
```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download the complete BankAccount and BankAccountTester classes.



### WORKED EXAMPLE 8.1

### Implementing a Menu Class



This Worked Example shows how to develop a class that displays a menu. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 8.1.



### VIDEO EXAMPLE 8.1

### Paying Off a Loan

When you take out a loan, the bank tells you how much you need to pay and for how long. Where do these numbers come from? This Video Example uses a Loan object to demonstrate how a loan is paid off. Go to [wiley.com/go/bjlo2videos](http://wiley.com/go/bjlo2videos) to view Video Example 8.1.



© Pavel Mitrofanov/  
iStockphoto.

## 8.8 Problem Solving: Tracing Objects

You have seen how the technique of hand-tracing is useful for understanding how a program works. When your program contains objects, it is useful to adapt the technique so that you gain a better understanding about object data and encapsulation.

Write the methods on the front of a card, and the instance variables on the back.

Use an index card or a sticky note for each object. On the front, write the methods that the object can execute. On the back, make a table for the values of the instance variables.

Here is a card for a CashRegister object:

|                                                                             |                         |
|-----------------------------------------------------------------------------|-------------------------|
| <b>CashRegister reg1</b><br>clear<br>addItem(price)<br>getTotal<br>getCount | <hr/> <hr/> <hr/> <hr/> |
| front                                                                       | back                    |

In a small way, this gives you a feel for encapsulation. An object is manipulated through its public interface (on the front of the card), and the instance variables are hidden in the back.

When an object is constructed, fill in the initial values of the instance variables:

|             |             |
|-------------|-------------|
| <hr/> <hr/> | <hr/> <hr/> |
| 0           | 0           |

Update the values of the instance variables when a mutator method is called.

Whenever a mutator method is executed, cross out the old values and write the new ones below. Here is what happens after a call to the addItem method:

|             |             |
|-------------|-------------|
| <hr/> <hr/> | <hr/> <hr/> |
| 0<br>1      | 0<br>19.95  |

If you have more than one object in your program, you will have multiple cards, one for each object:

|             |             |
|-------------|-------------|
| <hr/> <hr/> | <hr/> <hr/> |
| 0<br>1      | 0<br>19.95  |

|             |                     |
|-------------|---------------------|
| <hr/> <hr/> | <hr/> <hr/>         |
| 0<br>1<br>2 | 0<br>19.95<br>34.95 |

These diagrams are also useful when you design a class. Suppose you are asked to enhance the `CashRegister` class to compute the sales tax. Add a method `getSalesTax` to the front of the card. Now turn the card over, look over the instance variables, and ask yourself whether the object has sufficient information to compute the answer. Remember that each object is an autonomous unit. Any data value that can be used in a computation must be

- An instance variable.
- A method argument.
- A static variable (uncommon; see Section 8.10).

To compute the sales tax, we need to know the tax rate and the total of the taxable items. (Food items are usually not subject to sales tax.) We don't have that information available. Let us introduce additional instance variables for the tax rate and the taxable total. The tax rate can be set in the constructor (assuming it stays fixed for the lifetime of the object). When adding an item, we need to be told whether the item is taxable. If so, we add its price to the taxable total.

For example, consider the following statements.

```
CashRegister reg3 = new CashRegister(7.5); // 7.5 percent sales tax
reg3.addItem(3.95, false); // Not taxable
reg3.addItem(19.95, true); // Taxable
```

When you record the effect on a card, it looks like this:

| <code>reg3.itemCount</code> | <code>reg3.totalPrice</code> | <code>reg3.taxableTotal</code> | <code>reg3.taxRate</code> |
|-----------------------------|------------------------------|--------------------------------|---------------------------|
| 0                           | 0                            | 0                              | 7.5                       |
| 1                           | 3.95                         |                                |                           |
| 2                           | 23.90                        | 19.95                          |                           |



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download an enhanced `CashRegister` class that computes the sales tax.

With this information, it becomes easy to compute the tax. It is  $\text{taxableTotal} \times \text{taxRate} / 100$ . Tracing the object helped us understand the need for additional instance variables.



#### SELF CHECK

31. Consider a `Car` class that simulates fuel consumption in a car. We will assume a fixed efficiency (in miles per gallon) that is supplied in the constructor. There are methods for adding gas, driving a given distance, and checking the amount of gas left in the tank.

Make a card for a `Car` object, choosing suitable instance variables and showing their values after the object was constructed.

32. Trace the following method calls:

```
Car myCar = new Car(25);
myCar.addGas(20);
myCar.drive(100);
myCar.drive(200);
myCar.addGas(5);
```

33. Suppose you are asked to simulate the odometer of the car, by adding a method `getMilesDriven`. Add an instance variable to the object's card that is suitable for computing this method.
34. Trace the methods of Self Check 32, updating the instance variable that you added in Self Check 33.



© plusphoto/  
iStockphoto.

**Practice It** Now you can try these exercises at the end of the chapter: R8.13, R8.14, R8.15.



## Computing & Society 8.1 Open Source and Free Software

Most companies that produce software regard the source code as a trade secret. After all, if customers or competitors had access to the source code, they could study it and create similar programs without paying the original vendor. For the same reason, customers dislike secret source code. If a company goes out of business or decides to discontinue support for a computer program, its users are left stranded. They are unable to fix bugs or adapt the program to a new operating system. Fortunately, many software packages are distributed as "open source software", giving its users the right to see, modify, and redistribute the source code of a program.

Having access to source code is not sufficient to ensure that software serves the needs of its users. Some companies have created software that spies on users or restricts access to previously purchased books, music, or videos. If that software runs on a server or in an embedded device, the user cannot change its behavior. In the article <http://www.gnu.org/philosophy/free-software-even-more-important.en.html>, Richard Stallman, a famous computer scientist and winner of a MacArthur "genius" grant, describes the "free software movement" that champions the right of users to control what their software does. This is an ethical position that goes beyond using open source for reasons of convenience or cost savings.

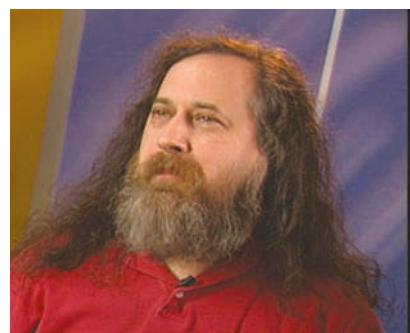
Stallman is the originator of the GNU project (<http://gnu.org/gnu/the-gnu-project.html>) that has produced an entirely free version of a UNIX-compatible operating system: the GNU operating system. All programs of the GNU project are licensed under the GNU General Public License (GNU GPL). The license allows you to make as many copies as you wish, make any modifications to the source, and redistribute the original and modified programs, charging nothing at all or whatever the market will bear. In return, you must agree that your modifications also fall under the license. You must give out the source code to any changes that you distribute, and anyone else can distribute them under the same conditions. The GNU GPL forms a social contract. Users of the software enjoy the freedom to use and modify the software, and in return they are obligated to share any improvements that they make available.

Some commercial software vendors have attacked the GPL as "viral" and "undermining the commercial software sector". Other companies have a more nuanced strategy, producing free or open source software, but charging for support or proprietary extensions. For example, the Java Development Kit is available under the GPL, but companies that need security updates for old versions or other support must pay Oracle.

Open source software sometimes lacks the polish of commercial software because many of the programmers are

volunteers who are interested in solving their own problems, not in making a product that is easy to use by everyone. Open source software has been particularly successful in areas that are of interest to programmers, such as the Linux kernel, Web servers, and programming tools.

The open source software community can be very competitive and creative. It is quite common to see several competing projects that take ideas from each other, all rapidly becoming more capable. Having many programmers involved, all reading the source code, often means that bugs tend to get squashed quickly. Eric Raymond describes open source development in his famous article "The Cathedral and the Bazaar" (<http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html>). He writes "Given enough eyeballs, all bugs are shallow".



Courtesy of Richard Stallman.

*Richard Stallman, a pioneer of the free source movement.*

## 8.9 Object References

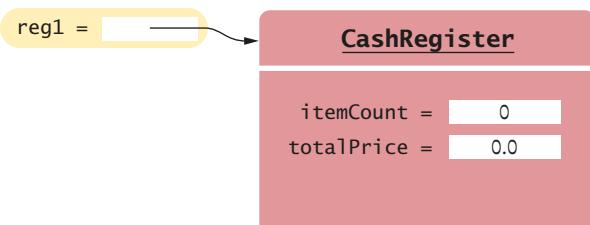
An object reference specifies the location of an object.

In Java, a variable whose type is a class does not actually hold an object. It merely holds the *memory location* of an object. The object itself is stored elsewhere—see Figure 8.

We use the technical term **object reference** to denote the memory location of an object. When a variable contains the memory location of an object, we say that it *refers* to an object. For example, after the statement

```
CashRegister reg1 = new CashRegister();
```

the variable `reg1` refers to the `CashRegister` object that the `new` operator constructed. Technically speaking, the `new` operator returned a reference to the new object, and that reference is stored in the `reg1` variable.



**Figure 8**  
An Object Variable Containing  
an Object Reference

### 8.9.1 Shared References

Multiple object variables can contain references to the same object.

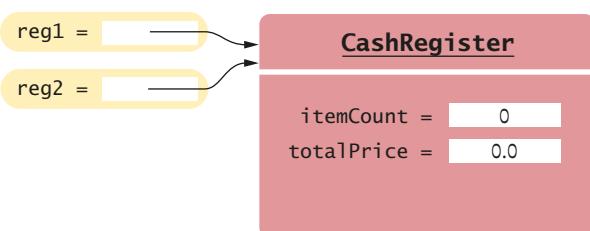
© Jacob Wackerhausen/  
iStockphoto.



You can have two (or more) object variables that store references to the same object, for example by assigning one to the other.

```
CashRegister reg2 = reg1;
```

Now you can access the same `CashRegister` object both as `reg1` and as `reg2`, as shown in Figure 9.



**Figure 9**  
Two Object Variables  
Referring to the Same Object

Primitive type variables store values. Object variables store references.

In this regard, object variables differ from variables for primitive types (numbers, characters, and `boolean` values). When you declare

```
int num1 = 0;
```

then the `num1` variable holds the number 0, not a reference to the number (see Figure 10).

num1 = [0]

**Figure 10** A Variable of Type `int` Stores a Number

You can see the difference between primitive type variables and object variables when you make a copy of a variable. When you copy a number, the original and the copy of the number are independent values. But when you copy an object reference, both the original and the copy are references to the same object.

Consider the following code, which copies a number and then changes the copy (see Figure 11):

```
int num1 = 0; ①
int num2 = num1; ②
num2++; ③
```

Now the variable `num1` contains the value 0, and `num2` contains 1.

Now consider the seemingly analogous code with `CashRegister` objects (see Figure 12):

```
CashRegister reg1 = new CashRegister(); ①
CashRegister reg2 = reg1; ②
reg2.addItem(2.95); ③
```

Because `reg1` and `reg2` refer to the same cash register after step ②, both variables now refer to a cash register with item count 1 and total price 2.95.

When copying an object reference, you have two references to the same object.

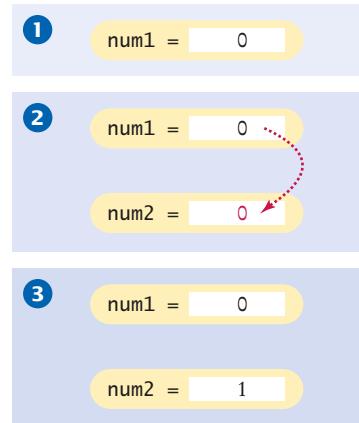


Figure 11 Copying Numbers

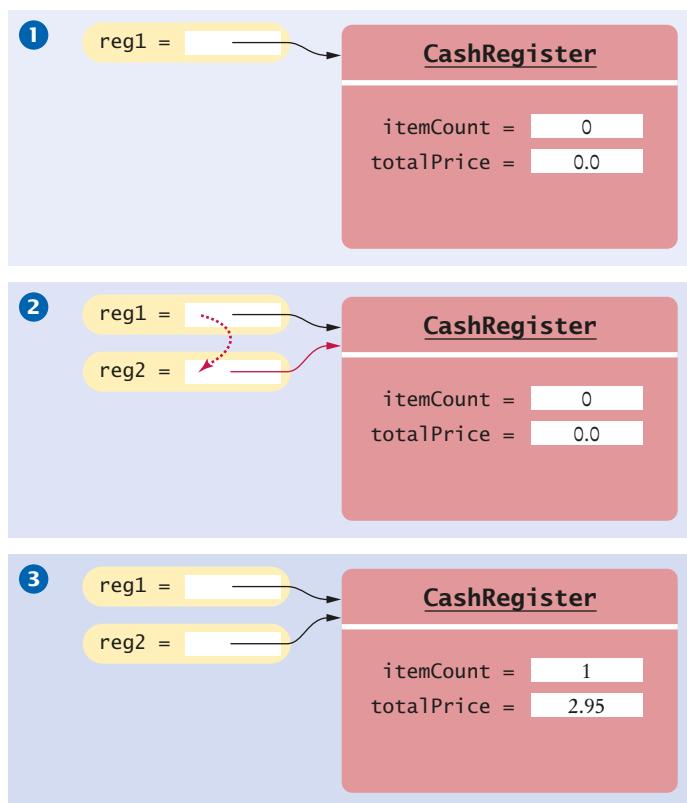


Figure 12 Copying Object References

There is a reason for the difference between numbers and objects. In the computer, each number requires a small amount of memory. But objects can be very large. It is far more efficient to manipulate only the memory location.

## 8.9.2 The null Reference

The `null` reference refers to no object.

An object reference can have the special value `null` if it refers to no object at all. It is common to use the `null` value to indicate that a value has never been set. For example,

```
String middleInitial = null; // No middle initial
```

You use the `==` operator (and not `equals`) to test whether an object reference is a `null` reference:

```
if (middleInitial == null)
{
 System.out.println(firstName + " " + lastName);
}
else
{
 System.out.println(firstName + " " + middleInitial + ". " + lastName);
}
```

Note that the `null` reference is not the same as the empty string `""`. The empty string is a valid string of length 0, whereas a `null` indicates that a `String` variable refers to no string at all.

It is an error to invoke a method on a `null` reference. For example,

```
CashRegister reg = null;
System.out.println(reg.getTotal()); // Error—cannot invoke a method on null
```

This code causes a “null pointer exception” at run time.

The `null` reference is the default value for an object reference that is contained inside another object or an array of objects. In order to avoid run-time errors, you need to replace these `null` references with references to actual objects.

For example, suppose you construct an array of bank accounts:

```
BankAccount[] accounts = new BankAccount[NACCOUNTS];
```

You now have an array filled with `null` references. If you want an array of actual bank accounts, you need to construct them:

```
for (int i = 0; i < accounts.length; i++)
{
 accounts[i] = new BankAccount();
}
```

## 8.9.3 The `this` Reference

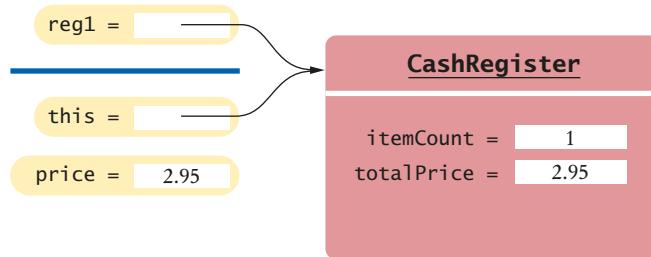
In a method, the `this` reference refers to the implicit parameter.

Every instance method receives the implicit parameter in a variable called `this`.

For example, consider the method call

```
reg1.addItem(2.95);
```

When the method is called, the parameter variable `this` refers to the same object as `reg1` (see Figure 13).

**Figure 13** The Implicit Parameter of a Method Call

You don't usually need to use the `this` reference, but you can. For example, you can write the `addItem` method like this:

```
void addItem(double price)
{
 this.itemCount++;
 this.totalPrice = this.totalPrice + price;
}
```

Some programmers like to use the `this` reference to make it clear that `itemCount` and `totalPrice` are instance variables and not local variables. You may want to try it out and see if you like that style.

There is another situation where the `this` reference can make your programs easier to read. Consider a constructor or instance method that calls another instance method *on the same object*. For example, the `CashRegister` constructor can call the `clear` method instead of duplicating its code:

```
public CashRegister()
{
 clear();
}
```

This call is easier to understand when you use the `this` reference:

```
public CashRegister()
{
 this.clear();
}
```

It is now more obvious that the method is invoked on the object that is being constructed.

Finally, some people like to use the `this` reference in constructors. Here is a typical example:

```
public class Student
{
 private int id;
 private String name;

 public Student(int id, String name)
 {
 this.id = id;
 this.name = name;
 }
}
```

The expression `id` refers to the parameter variable, and `this.id` to the instance variable. In general, if both a local variable and an instance variable have the same name, you can access the local variable by its name, and the instance variable with the `this` reference.

You can implement the constructor without using the `this` reference. Simply choose other names for the parameter variables:

```
public Student(int anId, String aName)
{
 id = anId;
 name = aName;
}
```

### SELF CHECK



- 35.** Suppose we have a variable  
`String greeting = "Hello";`  
 What is the effect of this statement?  
`String greeting2 = greeting;`
- 36.** After calling `String greeting3 = greeting2.toUpperCase()`, what are the contents of `greeting` and `greeting2`?
- 37.** What is the value of `s.length()` if `s` is
  - a. the empty string ""?
  - b. `null`?
- 38.** What is the type of `this` in the call `greeting.substring(1, 4)`?
- 39.** Supply a method `addItems(int quantity, double price)` in the `CashRegister` class to add multiple instances of the same item. Your implementation should repeatedly call the `addItem` method. Use the `this` reference.

### Practice It

Now you can try these exercises at the end of the chapter: R8.19, R8.20.

### Common Error 8.3



### Forgetting to Initialize Object References in a Constructor

Just as it is a common error to forget to initialize a local variable, it is easy to forget about instance variables. Every constructor needs to ensure that all instance variables are set to appropriate values.

If you do not initialize an instance variable, the Java compiler will initialize it for you. Numbers are initialized with 0, but object references—such as string variables—are set to the `null` reference.

Of course, 0 is often a convenient default for numbers. However, `null` is hardly ever a convenient default for objects. Consider this “lazy” constructor for a modified version of the `BankAccount` class:

```
public class BankAccount
{
 private double balance;
 private String owner;
 ...
 public BankAccount(double initialBalance)
 {
 balance = initialBalance;
 }
}
```

In this case, `balance` is initialized, but the `owner` variable is set to a `null` reference. This can be a problem—it is illegal to call methods on the `null` reference.

To avoid this problem, it is a good idea to initialize every instance variable:

```
public BankAccount(double initialBalance)
{
 balance = initialBalance;
 owner = "None";
}
```

### Special Topic 8.3



#### Calling One Constructor from Another

Consider the `BankAccount` class outlined in Section 8.6. It has two constructors: a constructor without arguments to initialize the balance with zero, and another constructor to supply an initial balance. Rather than explicitly setting the balance to zero, one constructor can call another constructor of the same class instead. There is a shorthand notation to achieve this result:

```
public class BankAccount
{
 public BankAccount(double initialBalance)
 {
 balance = initialBalance;
 }

 public BankAccount()
 {
 this(0);
 }
 ...
}
```

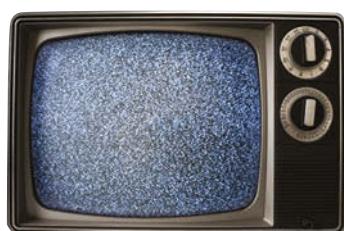
The command `this(0);` means “Call another constructor of this class and supply the value 0”. Such a call to another constructor can occur only as the *first line in a constructor*.

This syntax is a minor convenience. We will not use it in this book. Actually, the use of the reserved word `this` is a little confusing. Normally, `this` denotes a reference to the implicit parameter, but if `this` is followed by parentheses, it denotes a call to another constructor of this class.

## 8.10 Static Variables and Methods

A static variable belongs to the class, not to any object of the class.

Sometimes, a value properly belongs to a class, not to any object of the class. You use a **static variable** for this purpose. Here is a typical example: We want to assign bank account numbers sequentially. That is, we want the bank account constructor to construct the first account with number 1001, the next with number 1002, and so on. To solve this problem, we need to have a single value of `lastAssignedNumber` that is a property of the *class*, not any object of the class. Such a variable is called a static variable, because you declare it using the `static` reserved word.



*The reserved word `static` is a holdover from the C++ language. Its use in Java has no relationship to the normal use of the term.*

© Diane Diederich/  
iStockphoto.

```

public class BankAccount
{
 private double balance;
 private int accountNumber;
 private static int lastAssignedNumber = 1000;

 public BankAccount()
 {
 lastAssignedNumber++;
 accountNumber = lastAssignedNumber;
 }
 ...
}

```

Every `BankAccount` object has its own `balance` and `accountNumber` instance variables, but there is only a single copy of the `lastAssignedNumber` variable (see Figure 14). That variable is stored in a separate location, outside any `BankAccount` objects.

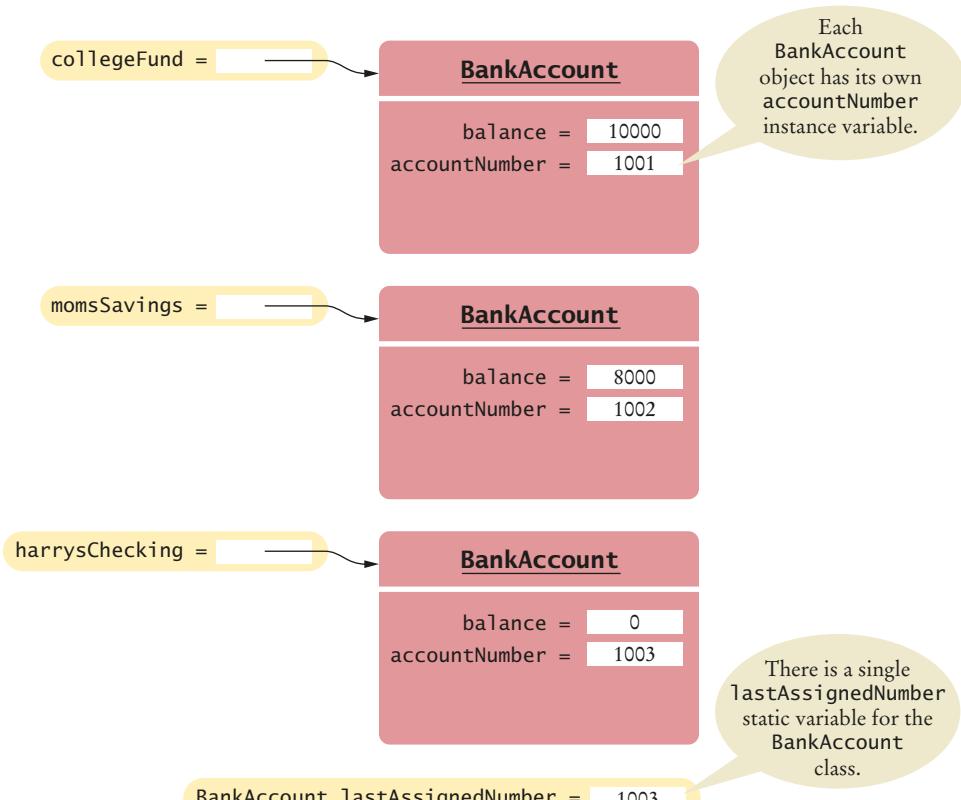
Like instance variables, static variables should always be declared as `private` to ensure that methods of other classes do not change their values. However, static *constants* may be either `private` or `public`. For example, the `BankAccount` class can define a public constant value, such as

```

public class BankAccount
{
 public static final double OVERDRAFT_FEE = 29.95;
 ...
}

```

Methods from any class can refer to such a constant as `BankAccount.OVERDRAFT_FEE`.



**Figure 14**  
A Static Variable and  
Instance Variables

A static method is not invoked on an object.

Sometimes a class defines methods that are not invoked on an object. Such a method is called a **static method**. A typical example of a static method is the `sqrt` method in the `Math` class. Because numbers aren't objects, you can't invoke methods on them. For example, if `x` is a number, then the call `x.sqrt()` is not legal in Java. Therefore, the `Math` class provides a static method that is invoked as `Math.sqrt(x)`. No object of the `Math` class is constructed. The `Math` qualifier simply tells the compiler where to find the `sqrt` method.

You can define your own static methods for use in other classes. Here is an example:

```
public class Financial
{
 /**
 * Computes a percentage of an amount.
 * @param percentage the percentage to apply
 * @param amount the amount to which the percentage is applied
 * @return the requested percentage of the amount
 */
 public static double percentOf(double percentage, double amount)
 {
 return (percentage / 100) * amount;
 }
}
```

When calling this method, supply the name of the class containing it:

```
double tax = Financial.percentOf(taxRate, total);
```

You had to use static methods in Chapter 5 before you knew how to implement your own objects. However, in object-oriented programming, static methods are not very common.

Nevertheless, the `main` method is always static. When the program starts, there aren't any objects. Therefore, the first method of a program must be a static method.

#### SELF CHECK



40. Name two static variables of the `System` class.
41. Name a static constant of the `Math` class.
42. The following method computes the average of an array of numbers:  

```
public static double average(double[] values)
```

Why should it not be defined as an instance method?
43. Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables `static`. Then `main` can call the other static methods, and all of them can access the static variables.  
 Will Harry's plan work? Is it a good idea?

**Practice It** Now you can try these exercises at the end of the chapter: E8.11, E8.12.

## 8.11 Problem Solving: Patterns for Object Data

When you design a class, you first consider the needs of the programmers who use the class. You provide the methods that the users of your class will call when they manipulate objects. When you implement the class, you need to come up with the instance variables for the class. It is not always obvious how to do this. Fortunately,



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program with static methods and variables.

there is a small set of recurring patterns that you can adapt when you design your own classes. We introduce these patterns in the following sections.

### 8.11.1 Keeping a Total

An instance variable for the total is updated in methods that increase or decrease the total amount.

Many classes need to keep track of a quantity that can go up or down as certain methods are called. Examples:

- A bank account has a balance that is increased by a deposit, decreased by a withdrawal.
- A cash register has a total that is increased when an item is added to the sale, cleared after the end of the sale.
- A car has gas in the tank, which is increased when fuel is added and decreased when the car drives.

In all of these cases, the implementation strategy is similar. Keep an instance variable that represents the current total. For example, for the cash register:

```
private double totalPrice;
```

Locate the methods that affect the total. There is usually a method to increase it by a given amount.

```
public void addItem(double price)
{
 totalPrice = totalPrice + price;
}
```

Depending on the nature of the class, there may be a method that reduces or clears the total. In the case of the cash register, there is a `clear` method:

```
public void clear()
{
 total = 0;
}
```

There is usually a method that yields the current total. It is easy to implement:

```
public double getTotal()
{
 return totalPrice;
}
```

All classes that manage a total follow the same basic pattern. Find the methods that affect the total and provide the appropriate code for increasing or decreasing it. Find the methods that report or use the total, and have those methods read the current total.

### 8.11.2 Counting Events

A counter that counts events is incremented in methods that correspond to the events.

You often need to count how often certain events occur in the life of an object. For example:

- In a cash register, you want to know how many items have been added in a sale.
- A bank account charges a fee for each transaction; you need to count them.

Keep a counter, such as

```
private int itemCount;
```

Increment the counter in those methods that correspond to the events that you want to count.

```
public void addItem(double price)
{
 totalPrice = totalPrice + price;
 itemCount++;
}
```

You may need to clear the counter, for example at the end of a sale or a statement period.

```
public void clear()
{
 total = 0;
 itemCount = 0;
}
```

There may or may not be a method that reports the count to the class user. The count may only be used to compute a fee or an average. Find out which methods in your class make use of the count, and read the current value in those methods.

### 8.11.3 Collecting Values

Some objects collect numbers, strings, or other objects. For example, each multiple-choice question has a number of choices. A cash register may need to store all prices of the current sale.

Use an array list or an array to store the values. (An array list is usually simpler because you won't need to track the number of values.) For example,

```
public class Question
{
 private ArrayList<String> choices;
 ...
}
```

In the constructor, initialize the instance variable to an empty collection:

```
public Question()
{
 choices = new ArrayList<String>();
}
```



*A shopping cart object needs to manage a collection of items.*

You need to supply some mechanism for adding values. It is common to provide a method for appending a value to the collection:

```
public void add(String choice)
{
 choices.add(choice);
}
```

The user of a Question object can call this method multiple times to add the various choices.

An object can collect other objects in an array or array list.

© paul prescott/iStockphoto.

### 8.11.4 Managing Properties of an Object

An object property can be accessed with a getter method and changed with a setter method.

A property is a value of an object that an object user can set and retrieve. For example, a Student object may have a name and an ID.

Provide an instance variable to store the property's value and methods to get and set it.

```
public class Student
{
 private String name;
 ...
 public String getName() { return name; }
 public void setName(String newName) { name = newName; }
 ...
}
```

It is common to add error checking to the setter method. For example, we may want to reject a blank name:

```
public void setName(String newName)
{
 if (newName.length() > 0) { name = newName; }
}
```

Some properties should not change after they have been set in the constructor. For example, a student's ID may be fixed (unlike the student's name, which may change). In that case, don't supply a setter method.

```
public class Student
{
 private int id;
 ...
 public Student(int anId) { id = anId; }
 public String getId() { return id; }
 // No setId method
 ...
}
```

### 8.11.5 Modeling Objects with Distinct States

If your object can have one of several states that affect the behavior, supply an instance variable for the current state.

Some objects have behavior that varies depending on what has happened in the past. For example, a Fish object may look for food when it is hungry and ignore food after it has eaten. Such an object would need to remember whether it has recently eaten.

Supply an instance variable that models the state, together with some constants for the state values:

```
public class Fish
{
 private int hungry;
 ...
 public static final int NOT_HUNGRY = 0;
 public static final int SOMEWHAT_HUNGRY = 1;
 public static final int VERY_HUNGRY = 2;
 ...
}
```



*If a fish is in a hungry state, its behavior changes.*

(Alternatively, you can use an enumeration—see Special Topic 3.4.)

Determine which methods change the state. In this example, a fish that has just eaten food, won't be hungry. But as the fish moves, it will get hungrier.

```
public void eat()
{
 hungry = NOT_HUNGRY;
 . .
}

public void move()
{
 .
 if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first.

```
public void move()
{
 if (hungry == VERY_HUNGRY)
 {
 Look for food.
 }
 .
}
```

### 8.11.6 Describing the Position of an Object

To model a moving object, you need to store and update its position.

Some objects move around during their lifetime, and they remember their current position. For example,

- A train drives along a track and keeps track of the distance from the terminus.
- A simulated bug living on a grid crawls from one grid location to the next, or makes 90 degree turns to the left or right.
- A cannonball is shot into the air, then descends as it is pulled by the gravitational force.

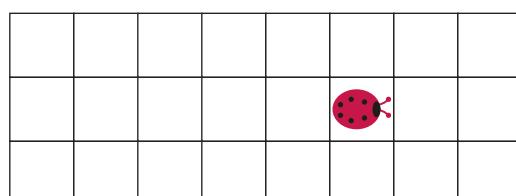
Such objects need to store their position. Depending on the nature of their movement, they may also need to store their orientation or velocity.

If the object moves along a line, you can represent the position as a distance from a fixed point.

```
private double distanceFromTerminus;
```

If the object moves in a grid, remember its current location and direction in the grid:

```
private int row;
private int column;
private int direction; // 0 = North, 1 = East, 2 = South, 3 = West
```



A bug in a grid needs to store its row, column, and direction.

When you model a physical object such as a cannonball, you need to track both the position and the velocity, possibly in two or three dimensions. Here we model a cannonball that is shot upward into the air:

```
private double zPosition;
private double zVelocity;
```

There will be methods that update the position. In the simplest case, you may be told by how much the object moves:

```
public void move(double distanceMoved)
{
 distanceFromTerminus = distanceFromTerminus + distanceMoved;
}
```

If the movement happens in a grid, you need to update the row or column, depending on the current orientation.

```
public void moveOneUnit()
{
 if (direction == NORTH) { row--; }
 else if (direction == EAST) { column++; }
 . .
}
```

Exercise P8.19 shows you how to update the position of a physical object with known velocity.

Whenever you have a moving object, keep in mind that your program will *simulate* the actual movement in some way. Find out the rules of that simulation, such as movement along a line or in a grid with integer coordinates. Those rules determine how to represent the current position. Then locate the methods that move the object, and update the positions according to the rules of the simulation.

### SELF CHECK



- 44.** Suppose we want to count the number of transactions in a bank account in a statement period, and we add a counter to the `BankAccount` class:
- ```
public class BankAccount
{
    private int transactionCount;
    . .
}
```
- In which methods does this counter need to be updated?
- 45.** In the example in Section 8.11.3, why is the `add` method required? That is, why can't the user of a `Question` object just call the `add` method of the `ArrayList<String>` class?
 - 46.** Suppose we want to enhance the `CashRegister` class in Section 8.6 to track the prices of all purchased items for printing a receipt. Which instance variable should you provide? Which methods should you modify?
 - 47.** Consider an `Employee` class with properties for tax ID number and salary. Which of these properties should have only a getter method, and which should have getter and setter methods?
 - 48.** Look at the `direction` instance variable in the bug example in Section 8.11.6. This is an example of which pattern?



Computing & Society 8.2 Electronic Voting Machines

In the 2000 presidential elections in the United States, votes were tallied by a variety of machines. Some machines processed cardboard ballots into which voters punched holes to indicate their choices (see below). When voters were not careful, remains of paper—the now infamous “chads”—were partially stuck in the punch cards, causing votes to be miscounted. A manual recount was necessary, but it was not carried out everywhere due to time constraints and procedural wrangling. The election was very close, and there remain doubts in the minds of many people whether the election outcome would have been different if the voting machines had accurately counted the intent of the voters.



Punch Card Ballot

Subsequently, voting machine manufacturers have argued that electronic voting machines would avoid the problems caused by punch cards or optically scanned forms. In an electronic voting machine, voters indicate their preferences by pressing buttons or touching icons on a computer screen. Typically, each voter is presented with a summary screen for review before casting the ballot. The process is very similar to using a bank's automated teller machine.

It seems plausible that these machines make it more likely that a vote is counted in the same way that the voter intends. However, there has been significant controversy surrounding some types of electronic voting machines. If a machine simply

records the votes and prints out the totals after the election has been completed, then how do you know that the machine worked correctly? Inside the machine is a computer that executes a program, and, as you may know from your own experience, programs can have bugs.

In fact, some electronic voting machines do have bugs. There have been isolated cases where machines reported tallies that were impossible. When a machine reports far more or far fewer votes than voters, then it is clear that it malfunctioned. Unfortunately, it is then impossible to find out the actual votes. Over time, one would expect these bugs to be fixed in the software. More insidiously, if the results are plausible, nobody may ever investigate.

Many computer scientists have spoken out on this issue and confirmed that it is impossible, with today's technology, to tell that software is error free and has not been tampered with. Many of them recommend that electronic voting machines should employ a *voter verifiable audit trail*. (A good source of information is <http://verifiedvoting.org>.) Typically, a voter-verifiable machine prints out a ballot. Each voter has a chance to review the printout, and then deposits it in an old-fashioned ballot box. If there is a problem with

the electronic equipment, the printouts can be scanned or counted by hand.

As this book is written, this concept is strongly resisted both by manufacturers of electronic voting machines and by their customers, the cities and counties that run elections. Manufacturers are reluctant to increase the cost of the machines because they may not be able to pass the cost increase on to their customers, who tend to have tight budgets. Election officials fear problems with malfunctioning printers, and some of them have publicly stated that they actually prefer equipment that eliminates bothersome recounts.

What do you think? You probably use an automated bank teller machine to get cash from your bank account. Do you review the paper record that the machine issues? Do you check your bank statement? Even if you don't, do you put your faith in other people who double-check their balances, so that the bank won't get away with widespread cheating?

Is the integrity of banking equipment more important or less important than that of voting machines? Won't every voting process have some room for error and fraud anyway? Is the added cost for equipment, paper, and staff time reasonable to combat

a potentially slight risk of malfunction and fraud? Computer scientists cannot answer these questions—an informed society must make these tradeoffs. But, like all professionals, they have an obligation to speak out and give accurate testimony about the capabilities and limitations of computing equipment.



Touch Screen Voting Machine

© Lisa F. Young/Stockphoto.

© Peter Nguyen/Stockphoto.



VIDEO EXAMPLE 8.2

Modeling a Robot Escaping from a Maze

In this Video Example, we will program classes that model a robot escaping from a maze. Go to wiley.com/go/bjlo2videos to view Video Example 8.2.

© Skip O'Donnell/
iStockphoto

8.12 Packages

A package is a set of related classes.

A Java program consists of a collection of classes. So far, most of your programs have consisted of a small number of classes. As programs get larger, however, simply distributing the classes over multiple files isn't enough. An additional structuring mechanism is needed.

In Java, packages provide this structuring mechanism. A Java **package** is a set of related classes. For example, the Java library consists of several hundred packages, some of which are listed in Table 1.

Table 1 Important Packages in the Java Library

Package	Purpose	Sample Class
java.lang	Language support	Math
java.util	Utilities	Random
java.io	Input and output	PrintStream
java.awt	Abstract Windowing Toolkit	Color
java.applet	Applets	Applet
java.net	Networking	Socket
java.sql	Database access through Structured Query Language	ResultSet
javax.swing	Swing user interface	JButton
org.w3c.dom	Document Object Model for XML documents	Document

8.12.1 Organizing Related Classes into Packages

To put one of your classes in a package, you must place a line

```
package packageName;
```

as the first instruction in the source file containing the class. A package name consists of one or more identifiers separated by periods. (See Section 8.12.3 for tips on constructing package names.)

For example, let's put the `Financial` class introduced in this chapter into a package named `com.horstmann.bigjava`.

In Java, related classes are grouped into packages.



© Don Wilkie/iStockphoto

The `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
    ...
}
```

In addition to the named packages (such as `java.util` or `com.horstmann.bigjava`), there is a special package, called the *default package*, which has no name. If you did not include any `package` statement at the top of your source file, its classes are placed in the default package.

8.12.2 Importing Packages

If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, `java.util.Scanner` refers to the `Scanner` class in the `java.util` package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

Naturally, that is somewhat inconvenient. For that reason, you usually import a name with an `import` statement:

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

Then you can refer to the class as `Scanner` without the package prefix.

You can import *all classes* of a package with an `import` statement that ends in `.*`. For example, you can use the statement

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

to import all classes from the `java.util` package. That statement lets you refer to classes like `Scanner` or `Random` without a `java.util` prefix.

However, you never need to import the classes in the `java.lang` package explicitly. That is the package containing the most basic Java classes, such as `Math` and `Object`. These classes are always available to you. In effect, an automatic `import java.lang.*;` statement has been placed into every source file.

Finally, you don't need to import other classes in the same package. For example, when you implement the class `homework1.Tester`, you don't need to import the class `homework1.Bank`. The compiler will find the `Bank` class without an `import` statement because it is located in the same package, `homework1`.

The `import` directive lets you refer to a class of a package by its class name, without the package prefix.

Syntax 8.4 Package Specification

Syntax `package packageName;`

`package com.horstmann.bigjava;`

The classes in this file
belong to this package.

A good choice for a package name
is a domain name in reverse.

8.12.3 Package Names

Placing related classes into a package is clearly a convenient mechanism to organize classes. However, there is a more important reason for packages: to avoid **name clashes**. In a large project, it is inevitable that two people will come up with the same name for the same concept. This even happens in the standard Java class library (which has now grown to thousands of classes). There is a class `Timer` in the `java.util` package and another class called `Timer` in the `javax.swing` package. You can still tell the Java compiler exactly which `Timer` class you need, simply by referring to them as `java.util.Timer` and `javax.swing.Timer`.

Of course, for the package-naming convention to work, there must be some way to ensure that package names are unique. It wouldn't be good if the car maker BMW placed all its Java code into the package `bmw`, and some other programmer (perhaps Britney M. Walters) had the same bright idea. To avoid this problem, the inventors of Java recommend that you use a package-naming scheme that takes advantage of the uniqueness of Internet domain names.

Use a domain
name in reverse
to construct an
unambiguous
package name.

For example, I have a domain name `horstmann.com`, and there is nobody else on the planet with the same domain name. (I was lucky that the domain name `horstmann.com` had not been taken by anyone else when I applied. If your name is Walters, you will sadly find that someone else beat you to `walters.com`.) To get a package name, turn the domain name around to produce a package name prefix, such as `com.horstmann`.

If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu`, then she can use a package name `edu.sjsu.cs.walters` for her own classes.

Some instructors will want you to place each of your assignments into a separate package, such as `homework1`, `homework2`, and so on. The reason is again to avoid name collision. You can have two classes, `homework1.Bank` and `homework2.Bank`, with slightly different properties.

8.12.4 Packages and Source Files

The path of a class
file must match its
package name.

A source file must be located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories. For example, the source files for classes in the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`. You place the subdirectory inside the *base directory* holding your program's files. For example, if you do your homework

assignment in a directory `/home/britney/hw8/problem1`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/britney/hw8/problem1/com/horstmann/bigjava`, as shown in Figure 15. (Here, we are using UNIX-style file names. Under Windows, you might use `c:\Users\Britney\hw8\problem1\com\horstmann\bigjava`.)

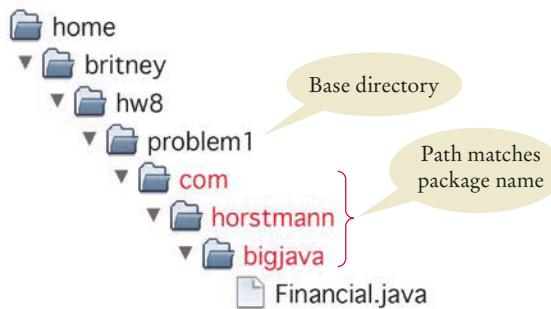


Figure 15 Base Directories and Subdirectories for Packages



SELF CHECK

49. Which of the following are packages?
 - a. `java`
 - b. `java.lang`
 - c. `java.util`
 - d. `java.lang.Math`
50. Is a Java program without `import` statements limited to using the default and `java.lang` packages?
51. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\Me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Practice It Now you can try these exercises at the end of the chapter: R8.21, E8.15, E8.16.

Common Error 8.4



Confusing Dots

In Java, the dot symbol (`.`) is used as a separator in the following situations:

- Between package names (`java.util`)
- Between package and class names (`homework1.Bank`)
- Between class and inner class names (`Ellipse2D.Double`)
- Between class and instance variable names (`Math.PI`)
- Between objects and methods (`account.getBalance()`)

When you see a long chain of dot-separated names, it can be a challenge to find out which part is the package name, which part is the class name, which part is an instance variable name, and which part is a method name. Consider

```
java.lang.System.out.println(x);
```

Because `println` is followed by an opening parenthesis, it must be a method name. Therefore, `out` must be either an object or a class with a static `println` method. (Of course, we know that `out` is an object reference of type `PrintStream`.) Again, it is not at all clear, without context, whether `System` is another object, with a public variable `out`, or a class with a static variable. Judging from the number of pages that the Java language specification devotes to this issue, even the compiler has trouble interpreting these dot-separated sequences of strings.

To avoid problems, it is helpful to adopt a strict coding style. If class names always start with an uppercase letter, and variable, method, and package names always start with a lowercase letter, then confusion can be avoided.

Special Topic 8.4



Package Access

If a class, instance variable, or method has no `public` or `private` modifier, then all methods of classes in the same package can access the feature. For example, if a class is declared as `public`, then all other classes in all packages can use it. But if a class is declared without an access modifier, then only the other classes in the *same* package can use it. Package access is a reasonable default for classes, but it is extremely unfortunate for instance variables.

An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

It is a common error to *forget* the reserved word `private`, thereby opening up a potential security hole. For example, at the time of this writing, the `Window` class in the `java.awt` package contained the following declaration:

```
public class Window extends Container
{
    String warningString;
    ...
}
```

There actually was no good reason to grant package access to the `warningString` instance variable—no other class accesses it.

Package access for instance variables is rarely useful and always a potential security risk. Most instance variables are given package access by accident because the programmer simply forgot the `private` reserved word. It is a good idea to get into the habit of scanning your instance variable declarations for missing `private` modifiers.

HOW TO 8.2



Programming with Packages

This How To explains in detail how to place your programs into packages.

Problem Statement Place each homework assignment into a separate package. That way, you can have classes with the same name but different implementations in separate packages (such as `homework1.problem1.Bank` and `homework1.problem2.Bank`).



© Don Wilkie/Stockphoto

Step 1

Come up with a package name.

Your instructor may give you a package name to use, such as `homework1.problem2`. Or, perhaps you want to use a package name that is unique to you. Start with your e-mail address, written

backwards. For example, `walters@cs.sjsu.edu` becomes `edu.sjsu.cs.walters`. Then add a sub-package that describes your project, such as `edu.sjsu.cs.walters.cs1project`.

Step 2 Pick a *base directory*.

The base directory is the directory that contains the directories for your various packages, for example, `/home/britney` or `c:\Users\Britney`.

Step 3 Make a subdirectory from the base directory that matches your package name.

The subdirectory must be contained in your base directory. Each segment must match a segment of the package name. For example,

```
mkdir -p /home/britney/homework1/problem2 (in UNIX)
or
mkdir /s c:\Users\Britney\homework1\problem2 (in Windows)
```

Step 4 Place your source files into the package subdirectory.

For example, if your homework consists of the files `Tester.java` and `Bank.java`, then you place them into

```
/home/britney/homework1/problem2/Tester.java
/home/britney/homework1/problem2/Bank.java
or
c:\Users\Britney\homework1\problem2\Tester.java
c:\Users\Britney\homework1\problem2\Bank.java
```

Step 5 Use the package statement in each source file.

The first noncomment line of each file must be a package statement that lists the name of the package, such as

```
package homework1.problem2;
```

Step 6 Compile your source files from the *base directory*.

Change to the base directory (from Step 2) to compile your files. For example,

```
cd /home/britney
javac homework1/problem2/Tester.java
or
c:
cd \Users\Britney
javac homework1\problem2\Tester.java
```

Note that the Java compiler needs the *source file name and not the class name*. That is, you need to supply file separators (/ on UNIX, \ on Windows) and a file extension (.java).

Step 7 Run your program from the *base directory*.

Unlike the Java compiler, the Java interpreter needs the *class name (not a file name) of the class containing the main method*. That is, use periods as package separators, and don't use a file extension. For example,

```
cd /home/britney
java homework1.problem2.Tester
or
c:
cd \Users\Britney
java homework1.problem2.Tester
```

CHAPTER SUMMARY

Understand the concepts of classes, objects, and encapsulation.



- A class describes a set of objects with the same behavior.
- Every class has a public interface: a collection of methods through which the objects of the class can be manipulated.
- Encapsulation is the act of providing a public interface and hiding the implementation details.
- Encapsulation enables changes in the implementation without affecting users of a class.



Understand instance variables and method implementations of a simple class.

- The methods of a class define the behavior of its objects.
- An object's instance variables represent the state of the object.
- Each object of a class has its own set of instance variables.
- An instance method can access the instance variables of the object on which it acts.
- A private instance variable can only be accessed by the methods of its own class.



Write method headers that describe the public interface of a class.



- You can use method headers and method comments to specify the public interface of a class.
- A mutator method changes the object on which it operates.
- An accessor method does not change the object on which it operates.

Choose an appropriate data representation for a class.



- Each accessor method must either retrieve a stored value or compute the result.
- Commonly, there is more than one way of representing the data of an object, and you must make a choice.
- Be sure that your data representation supports method calls in any order.

Provide the implementation of instance methods for a class.

- The object on which an instance method is applied is the implicit parameter.
- Explicit parameters of a method are listed in the method declaration.



Design and implement constructors.



- A constructor initializes the instance variables of an object.
- A constructor is invoked when an object is created with the new operator.
- The name of a constructor is the same as the class name.

- A class can have multiple constructors.
- The compiler picks the constructor that matches the construction arguments.
- By default, numbers are initialized as 0, Booleans as `false`, and object references as `null`.
- If you do not provide a constructor, a constructor with no arguments is generated.

Write tests that verify that a class works correctly.



- A unit test verifies that a class works correctly in isolation, outside a complete program.
- To test a class, use an environment for interactive testing, or write a tester class to execute test instructions.
- Determining the expected result in advance is an important part of testing.

Use the technique of object tracing for visualizing object behavior.

- Write the methods on the front of a card, and the instance variables on the back.
- Update the values of the instance variables when a mutator method is called.

Describe the behavior of object references.



- An object reference specifies the location of an object.
- Multiple object variables can contain references to the same object.
- Primitive type variables store values. Object variables store references.
- When copying an object reference, you have two references to the same object.
- The `null` reference refers to no object.
- In a method, the `this` reference refers to the implicit parameter.

Understand the behavior of static variables and methods.



- A static variable belongs to the class, not to any object of the class.
- A static method is not invoked on an object.

Use patterns to design the data representation of a class.



- An instance variable for the total is updated in methods that increase or decrease the total amount.
- A counter that counts events is incremented in methods that correspond to the events.
- An object can collect other objects in an array or array list.
- An object property can be accessed with a getter method and changed with a setter method.
- If your object can have one of several states that affect the behavior, supply an instance variable for the current state.
- To model a moving object, you need to store and update its position.



Use packages to organize sets of related classes.

- A package is a set of related classes.
- The `import` directive lets you refer to a class of a package by its class name, without the package prefix.
- Use a domain name in reverse to construct an unambiguous package name.
- The path of a class file must match its package name.
- An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

**REVIEW EXERCISES**

■ **R8.1** What is encapsulation? Why is it useful?

■ **R8.2** What values are returned by the calls `reg1.getCount()`, `reg1.getTotal()`, `reg2.getCount()`, and `reg2.getTotal()` after these statements?

```
CashRegister reg1 = new CashRegister();
reg1.addItem(3.25);
reg1.addItem(1.95);
CashRegister reg2 = new CashRegister();
reg2.addItem(3.25);
reg2.clear();
```

■ **R8.3** Consider the `Menu` class in Worked Example 8.1. What is displayed when the following calls are executed?

```
Menu simpleMenu = new Menu();
simpleMenu.addOption("Ok");
simpleMenu.addOption("Cancel");
int response = simpleMenu.getInput();
```

■ **R8.4** What is the *public interface* of a class? How does it differ from the *implementation* of a class?

■ ■ **R8.5** Consider the data representation of a cash register that keeps track of sales tax in Section 8.8. Instead of tracking the taxable total, track the total sales tax. Redo the walkthrough with this change.

■ ■ ■ **R8.6** Suppose the `CashRegister` needs to support a method `void undo()` that undoes the addition of the preceding item. This enables a cashier to quickly undo a mistake. What instance variables should you add to the `CashRegister` class to support this modification?

■ **R8.7** What is an instance method, and how does it differ from a static method?

■ **R8.8** What is a mutator method? What is an accessor method?

■ **R8.9** What is an implicit parameter? How does it differ from an explicit parameter?

- **R8.10** How many implicit parameters can an instance method have? How many implicit parameters can a static method have? How many explicit parameters can an instance method have?
- **R8.11** What is a constructor?
- **R8.12** How many constructors can a class have? Can you have a class with no constructors? If a class has more than one constructor, which of them gets called?
- **R8.13** Using the object tracing technique described in Section 8.8, trace the program at the end of Section 8.7.
- **R8.14** Using the object tracing technique described in Section 8.8, trace the program in How To 8.1 on page 395.
- **R8.15** Design a modification of the `BankAccount` class in How To 8.1 in which the first five transactions per month are free and a \$1 fee is charged for every additional transaction. Provide a method that deducts the fee at the end of a month. What additional instance variables do you need? Using the object tracing technique described in Section 8.8, trace a scenario that shows how the fees are computed over two months.
- **R8.16** Instance variables are “hidden” by declaring them as `private`, but they aren’t hidden very well at all. Anyone can read the class declaration. Explain to what extent the `private` reserved word hides the `private` implementation of a class.
- **R8.17** You can read the `itemCount` instance variable of the `CashRegister` class with the `getCount` accessor method. Should there be a `setCount` mutator method to change it? Explain why or why not.
- **R8.18** In a static method, it is easy to differentiate between calls to instance methods and calls to static methods. How do you tell them apart? Why is it not as easy for methods that are called from an instance method?
- **R8.19** What is the `this` reference? Why would you use it?
- **R8.20** What is the difference between the number zero, the `null` reference, the value `false`, and the empty string?
- **R8.21** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `CaesarCipher.java` from Section 7.3 to avoid `import` statements.
- **R8.22** What is the default package? Have you used it before this chapter in your programming?

PRACTICE EXERCISES

- **E8.1** We want to add a button to the tally counter in Section 8.2 that allows an operator to undo an accidental button click. Provide a method

```
public void undo()
```

that simulates such a button. As an added precaution, make sure that the operator cannot click the undo button more often than the count button.

- **E8.2** Simulate a tally counter that can be used to admit a limited number of people. First, the limit is set with a call

```
public void setLimit(int maximum)
```

If the count button was clicked more often than the limit, simulate an alarm by printing out a message "Limit exceeded".

- ■ **E8.3** Simulate a circuit for controlling a hallway light that has switches at both ends of the hallway. Each switch can be up or down, and the light can be on or off. Toggling either switch turns the lamp on or off. Provide methods

```
public int getFirstSwitchState() // 0 for down, 1 for up
public int getSecondSwitchState()
public int getLampState() // 0 for off, 1 for on
public void toggleFirstSwitch()
public void toggleSecondSwitch()
```

- **E8.4** Change the public interface of the circuit class of Exercise E8.3 so that it has the following methods:

```
public int getSwitchState(int switchNum)
public int getLampState()
public void toggleSwitch(int switchNum)
```

- ■ ■ **E8.5** Reimplement the `Menu` class of Worked Example 8.1 so that it stores all menu items in one long string. *Hint:* Keep a separate counter for the number of options. When a new option is added, append the option count, the option, and a newline character.

- ■ **E8.6** Implement a class `Address`. An address has a house number, a street, an optional apartment number, a city, a state, and a postal code. Supply two constructors: one with an apartment number and one without. Supply a `print` method that prints the address with the street on one line and the city, state, and zip code on the next line. Supply a method `public boolean comesBefore(Address other)` that tests whether this address comes before another when the addresses are compared by postal code.

- **E8.7** Implement a class `Student`. For the purpose of this exercise, a student has a name and a total quiz score. Supply an appropriate constructor and methods `getName()`, `addQuiz(int score)`, `getTotalScore()`, and `getAverageScore()`. To compute the latter, you also need to store the *number of quizzes* that the student took.

- ■ **E8.8** Modify the `Student` class of Exercise E8.7 to compute grade point averages. Methods are needed to add a grade and get the current GPA. Specify grades as elements of a class `Grade`. Supply a constructor that constructs a grade from a string, such as "B+". You will also need a method that translates grades into their numeric values (for example, "B+" becomes 3.3).

- ■ **E8.9** Write a class `Bug` that models a bug moving along a horizontal line. The bug moves either to the right or left. Initially, the bug moves to the right, but it can turn to change its direction. In each move, its position changes by one unit in the current direction. Provide a constructor

```
public Bug(int initialPosition)
```

and methods

- `public void turn()`
- `public void move()`
- `public int getPosition()`

Sample usage:

```
Bug bugsy = new Bug(10);
bugsy.move(); // Now the position is 11
bugsy.turn();
bugsy.move(); // Now the position is 10
```

Your `main` method should construct a bug, make it move and turn a few times, and print the actual and expected positions.

- E8.10** Implement a class `Moth` that models a moth flying in a straight line. The moth has a position, the distance from a fixed origin. When the moth moves toward a point of light, its new position is halfway between its old position and the position of the light source. Supply a constructor

```
public Moth(double initialPosition)
```

and methods

- public void `moveToLight(double lightPosition)`
- public void `getPosition()`

Your `main` method should construct a moth, move it toward a couple of light sources, and check that the moth's position is as expected.

- E8.11** Write static methods

- public static double `sphereVolume(double r)`
- public static double `sphereSurface(double r)`
- public static double `cylinderVolume(double r, double h)`
- public static double `cylinderSurface(double r, double h)`
- public static double `coneVolume(double r, double h)`
- public static double `coneSurface(double r, double h)`

that compute the volume and surface area of a sphere with a radius `r`, a cylinder with a circular base with radius `r` and height `h`, and a cone with a circular base with radius `r` and height `h`. Place them into a class `Geometry`. Then write a program that prompts the user for the values of `r` and `h`, calls the six methods, and prints the results.

- E8.12** Solve Exercise E8.11 by implementing classes `Sphere`, `Cylinder`, and `Cone`. Which approach is more object-oriented?

- E8.13** Implement a class `LoginForm` that simulates a login form that you find on many web pages. Supply methods

```
public void input(String text)
public void click(String button)
public boolean loggedIn()
```

The first input is the user name, the second input is the password. The `click` method can be called with arguments "Submit" and "Reset". Once a user has been successfully logged in, by supplying the user name, password, and clicking on the submit button, the `loggedIn` method returns true and further input has no effect. When a user tries to log in with an invalid user name and password, the form is reset.

Supply a constructor with the expected user name and password.

- E8.14** Implement a class `Robot` that simulates a robot wandering on an infinite plane. The robot is located at a point with integer coordinates and faces north, east, south, or west.

Supply methods

```
public void turnLeft()
public void turnRight()
public void move()
public Point getLocation()
public String getDirection()
```

The `turnLeft` and `turnRight` methods change the direction but not the location. The `move` method moves the robot by one unit in the direction it is facing. The `getDirection` method returns a string "N", "E", "S", or "W".

- ■ **E8.15** Implement a `CashRegister` class and place the class into a package called `money`. Keep the `CashRegisterTester` class in the default package.
- **E8.16** Place a `BankAccount` class in a package whose name is derived from your e-mail address, as described in Section 8.12.3. Keep the `BankAccountTester` class in the default package.

PROGRAMMING PROJECTS

- ■ **P8.1** A microwave control panel has four buttons: one for increasing the time by 30 seconds, one for switching between power levels 1 and 2, a reset button, and a start button. Implement a class that simulates the microwave, with a method for each button. The method for the start button should print a message "Cooking for ... seconds at level ...".
- **P8.2** A `Person` has a name (just a first name for simplicity) and friends. Store the names of the friends in a string, separated by spaces. Provide a constructor that constructs a person with a given name and no friends. Provide methods

```
public void befriend(Person p)
public void unfriend(Person p)
public String getFriendNames()
```

- **P8.3** Add a method

```
public int getFriendCount()
```

to the `Person` class of Exercise P8.2.

- **P8.4** Write a class `Battery` that models a rechargeable battery. A battery has a constructor

```
public Battery(double capacity)
```

where `capacity` is a value measured in milliampere hours. A typical AA battery has a capacity of 2000 to 3000 mAh. The method

```
public void drain(double amount)
```

drains the capacity of the battery by the given amount. The method

```
public void charge()
```

charges the battery to its original capacity.

The method

```
public double getRemainingCapacity()
```

gets the remaining capacity of the battery.



- **P8.5** Implement a class `SodaCan` with methods `getSurfaceArea()` and `getVolume()`. In the constructor, supply the height and radius of the can.
- **P8.6** Implement a class `Car` with the following properties. A car has a certain fuel efficiency (measured in miles/gallon) and a certain amount of fuel in the gas tank. The efficiency is specified in the constructor, and the initial fuel level is 0. Supply a method `drive` that simulates driving the car for a certain distance, reducing the fuel level in the gas tank, and methods `getGasLevel`, to return the current fuel level, and `addGas`, to tank up. Sample usage:

```
Car myHybrid = new Car(50); // 50 miles per gallon
myHybrid.addGas(20); // Tank 20 gallons
myHybrid.drive(100); // Drive 100 miles
System.out.println(myHybrid.getGasLevel()); // Print fuel remaining
```

- **P8.7** Declare a class `ComboLock` that works like the combination lock in a gym locker, as shown here. The lock is constructed with a combination—three numbers between 0 and 39. The `reset` method resets the dial so that it points to 0. The `turnLeft` and `turnRight` methods turn the dial by a given number of ticks to the left or right. The `open` method attempts to open the lock. The lock opens if the user first turned it right to the first number in the combination, then left to the second, and then right to the third.

```
public class ComboLock {
    ...
    public ComboLock(int secret1, int secret2, int secret3) { . . . }
    public void reset() { . . . }
    public void turnLeft(int ticks) { . . . }
    public void turnRight(int ticks) { . . . }
    public boolean open() { . . . }
}
```

- **P8.8** Implement a `VotingMachine` class that can be used for a simple election. Have methods to clear the machine state, to vote for a Democrat, to vote for a Republican, and to get the tallies for both parties.

- **P8.9** Provide a class for authoring a simple letter. In the constructor, supply the names of the sender and the recipient:

```
public Letter(String from, String to)
```

Supply a method

```
public void addLine(String line)
```

to add a line of text to the body of the letter. Supply a method

```
public String getText()
```

that returns the entire text of the letter. The text has the form:

```
Dear recipient name:  
blank line  
first line of the body  
second line of the body  
...  
last line of the body  
blank line  
Sincerely,  
blank line  
sender name
```

Also supply a `main` method that prints this letter.

Dear John:

I am sorry we must part.
I wish you all the best.

Sincerely,

Mary

Construct an object of the `Letter` class and call `addLine` twice.

■■ Business P8.10 Reimplement the `CashRegister` class so that it keeps track of the price of each added item in an `ArrayList<Double>`. Remove the `itemCount` and `totalPrice` instance variables. Reimplement the `clear`, `addItem`, `getTotal`, and `getCount` methods. Add a method `displayAll` that displays the prices of all items in the current sale.

■■ Business P8.11 Reimplement the `CashRegister` class so that it keeps track of the total price as an integer: the total cents of the price. For example, instead of storing 17.29, store the integer 1729. Such an implementation is commonly used because it avoids the accumulation of roundoff errors. Do not change the public interface of the class.

■■ Business P8.12 After closing time, the store manager would like to know how much business was transacted during the day. Modify the `CashRegister` class to enable this functionality. Supply methods `getSalesTotal` and `getSalesCount` to get the total amount of all sales and the number of sales. Supply a method `resetSales` that resets any counters and totals so that the next day's sales start from zero.

■■ Business P8.13 Implement a class `Portfolio`. This class has two objects, `checking` and `savings`, of the type `BankAccount` that was developed in How To 8.1 (`ch08/how_to_1/BankAccount.java` in the companion code for this book). Implement four methods:

- `public void deposit(double amount, String account)`
- `public void withdraw(double amount, String account)`
- `public void transfer(double amount, String account)`
- `public double getBalance(String account)`

Here the account string is "S" or "C". For the deposit or withdrawal, it indicates which account is affected. For a transfer, it indicates the account from which the money is taken; the money is automatically transferred to the other account.

■■ Business P8.14 Design and implement a class `Country` that stores the name of the country, its population, and its area. Then write a program that reads in a set of countries and prints

- The country with the largest area.
- The country with the largest population.
- The country with the largest population density (people per square kilometer (or mile)).

■■ Business P8.15 Design a class `Message` that models an e-mail message. A message has a recipient, a sender, and a message text. Support the following methods:

- A constructor that takes the sender and recipient
- A method `append` that appends a line of text to the message body
- A method `toString` that makes the message into one long string like this: "From:
Harry Morgan%nTo: Rudolf Reindeer%n . . ."

Write a program that uses this class to make a message and print it.

■■ Business P8.16 Design a class `Mailbox` that stores e-mail messages, using the `Message` class of Exercise P8.15. Implement the following methods:

- `public void addMessage(Message m)`
- `public Message getMessage(int i)`
- `public void removeMessage(int i)`

■■ Business P8.17 Design a `Customer` class to handle a customer loyalty marketing campaign. After accumulating \$100 in purchases, the customer receives a \$10 discount on the next purchase. Provide methods

- `void makePurchase(double amount)`
- `boolean discountReached()`

Provide a test program and test a scenario in which a customer has earned a discount and then made over \$90, but less than \$100 in purchases. This should not result in a second discount. Then add another purchase that results in the second discount.

■■■ Business P8.18 The Downtown Marketing Association wants to promote downtown shopping with a loyalty program similar to the one in Exercise P8.17. Shops are identified by a number between 1 and 20. Add a new parameter variable to the `makePurchase` method that indicates the shop. The discount is awarded if a customer makes purchases in at least three different shops, spending a total of \$100 or more.



© ThreeJays/Stockphoto.

■■■ Science P8.19 Design a class `Cannonball` to model a cannonball that is fired into the air. A ball has

- An x - and a y -position.
- An x - and a y -velocity.

Supply the following methods:

- A constructor with an x -position (the y -position is initially 0)
- A method `move(double sec)` that moves the ball to the next position (First compute the distance traveled in sec seconds, using the current velocities, then update the x - and y -positions; then update the y -velocity by taking into account the gravitational acceleration of -9.81 m/s^2 ; the x -velocity is unchanged.)
- Methods `getX` and `getY` that get the current location of the cannonball
- A method `shoot` whose arguments are the angle α and initial velocity v (Compute the x -velocity as $v \cos \alpha$ and the y -velocity as $v \sin \alpha$; then keep calling `move` with a time interval of 0.1 seconds until the y -position is 0; call `getX` and `getY` after every move and display the position.)

Use this class in a program that prompts the user for the starting angle and the initial velocity. Then call `shoot`.

■■ Science P8.20 The colored bands on the top-most resistor shown in the photo below indicate a resistance of $6.2 \text{ k}\Omega \pm 5$ percent. The resistor tolerance of ± 5 percent indicates the acceptable variation in the resistance. A $6.2 \text{ k}\Omega \pm 5$ percent resistor could have a resistance as small as $5.89 \text{ k}\Omega$ or as large as $6.51 \text{ k}\Omega$. We say that $6.2 \text{ k}\Omega$ is the *nominal value* of the resistance and that the actual value of the resistance can be any value between $5.89 \text{ k}\Omega$ and $6.51 \text{ k}\Omega$.

Write a program that represents a resistor as a class. Provide a single constructor that accepts values for the nominal resistance and tolerance and then determines the actual value randomly. The class should provide public methods to get the nominal resistance, tolerance, and the actual resistance.

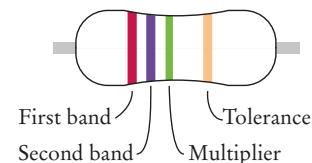
Write a `main` method for the program that demonstrates that the class works properly by displaying actual resistances for ten $330 \Omega \pm 10$ percent resistors.

- Science P8.21** In the `Resistor` class from Exercise P8.20, supply a method that returns a description of the “color bands” for the resistance and tolerance. A resistor has four color bands:

- The first band is the first significant digit of the resistance value.
- The second band is the second significant digit of the resistance value.
- The third band is the decimal multiplier.
- The fourth band indicates the tolerance.



© Maria Toutoudaki/iStockphoto.



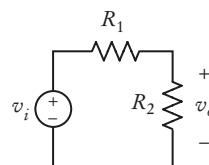
Color	Digit	Multiplier	Tolerance
Black	0	$\times 10^0$	—
Brown	1	$\times 10^1$	$\pm 1\%$
Red	2	$\times 10^2$	$\pm 2\%$
Orange	3	$\times 10^3$	—
Yellow	4	$\times 10^4$	—
Green	5	$\times 10^5$	$\pm 0.5\%$
Blue	6	$\times 10^6$	$\pm 0.25\%$
Violet	7	$\times 10^7$	$\pm 0.1\%$
Gray	8	$\times 10^8$	$\pm 0.05\%$
White	9	$\times 10^9$	—
Gold	—	$\times 10^{-1}$	$\pm 5\%$
Silver	—	$\times 10^{-2}$	$\pm 10\%$
None	—	—	$\pm 20\%$

For example (using the values from the table as a key), a resistor with red, violet, green, and gold bands (left to right) will have 2 as the first digit, 7 as the second digit, a multiplier of 10^5 , and a tolerance of ± 5 percent, for a resistance of $2,700 \text{ k}\Omega$, plus or minus 5 percent.

Science P8.22 The figure below shows a frequently used electric circuit called a “voltage divider”. The input to the circuit is the voltage v_i . The output is the voltage v_o . The output of a voltage divider is proportional to the input, and the constant of proportionality is called the “gain” of the circuit. The voltage divider is represented by the equation

$$G = \frac{v_o}{v_i} = \frac{R_2}{R_1 + R_2}$$

where G is the gain and R_1 and R_2 are the resistances of the two resistors that comprise the voltage divider.



Manufacturing variations cause the actual resistance values to deviate from the nominal values, as described in Exercise P8.20. In turn, variations in the resistance values cause variations in the values of the gain of the voltage divider. We calculate the *nominal value of the gain* using the nominal resistance values and the *actual value of the gain* using actual resistance values.

Write a program that contains two classes, `VoltageDivider` and `Resistor`. The `Resistor` class is described in Exercise P8.20. The `VoltageDivider` class should have two instance variables that are objects of the `Resistor` class. Provide a single constructor that accepts two `Resistor` objects, nominal values for their resistances, and the resistor tolerance. The class should provide public methods to get the nominal and actual values of the voltage divider’s gain.

Write a `main` method for the program that demonstrates that the class works properly by displaying nominal and actual gain for ten voltage dividers each consisting of 5 percent resistors having nominal values $R_1 = 250 \Omega$ and $R_2 = 750 \Omega$.

ANSWERS TO SELF-CHECK QUESTIONS

1. No—the object "Hello, World" belongs to the `String` class, and the `String` class has no `println` method.
2. Through the `substring` and `charAt` methods.
3. As an `ArrayList<Character>`. As a `char` array.
4. None. The methods will have the same effect, and your code could not have manipulated `String` objects in any other way.
5.

```
public void reset()
{
    value = 0;
}
```
6.

```
public int getValue()
{
```
7. None—the public interface has not changed.
8. You cannot access the instance variables directly. You must use the methods provided by the `Clock` class.
9. 2 1.90
10. There is no method named `getAmountDue`.
11. `public int getDollars();`
12. `length`, `substring`. In fact, all methods of the `String` class are accessors.
13. A mutator. Getting the next number removes it from the input, thereby modifying it. Not

convinced? Consider what happens if you call the `nextInt` method twice. You will usually get two different numbers. But if you call an accessor twice on an object (without a mutation between the two calls), you are sure to get the same result.

14.

```
14. /**
   This class models a tally counter.
 */
public class Counter
{
    private int value;

    /**
     Gets the current value of this counter.
     @return the current value
    */
    public int getValue()
    {
        return value;
    }

    /**
     Advances the value of this counter by 1.
    */
    public void count()
    {
        value = value + 1;
    }
}
```

15. The code tries to access a private instance variable.

16. (1) `int hours; // Between 1 and 12`
`int minutes; // Between 0 and 59`
`boolean pm; // True for P.M., false for A.M.`

(2) `int hours; // Military time, between 0 and 23`
`int minutes; // Between 0 and 59`

(3) `int totalMinutes // Between 0 and 60 * 24 - 1`

17. They need not change their programs at all because the public interface has not changed. They need to recompile with the new version of the `Time` class.

18. (1) `String letterGrade; // "A+", "B"`
(2) `double numberGrade; // 4.3, 3.0`

19. 2 1.85 1 1.90

20. `public int getDollars()`
{
 int dollars = (int) totalPrice;
 // Truncates cents
 return dollars;
}

21. Three parameters: two explicit parameters of type `int`, and one implicit parameter of type `String`.

22. One parameter: the implicit parameter of type `String`. The method has no explicit parameters.

23. "Morgan, Harry"

24. `public Person() { name = "unknown"; }`

25. A constructor is generated that has the same effect as the constructor provided in this section. It sets both instance variables to zero.

26. `public Item()`

```
{
    price = 0;
    description = "";
}
```

The `price` instance variable need not be initialized because it is set to zero by default, but it is clearer to initialize it explicitly.

27. (a) `Item(String)` (b) `Item(double)`
(c) `Item(String, double)` (d) `Item()`
(e) No constructor has been called.

28. Add these lines:

```
register1.clear();
System.out.println(register1.getCount());
System.out.println("Expected: 0");
System.out.printf("%.2f%n",
    register1.getTotal());
System.out.println("Expected: 0.00");
```

29. 1, 0

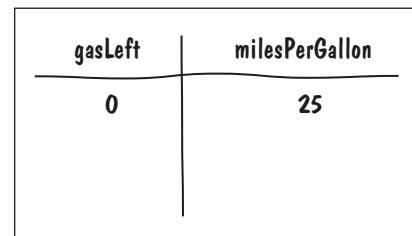
30. These environments allow you to call methods on an object without creating a `main` method.

31.

Car myCar

```
Car(mpg)
addGas(amount)
drive(distance)
getGasLeft
```

front



back

32.

gasLeft	milesPerGallon
0	
20	
16	
8	
13	25

33.

gasLeft	milesPerGallon	totalMiles
0	25	0

34.

gasLeft	milesPerGallon	totalMiles
0	25	0
20		
16		100
8		300
13		

35. Both greeting and greeting2 refer to the same string "Hello".

36. They both still refer to the string "Hello". The toUpperCase method computes the string "HELLO", but it is not a mutator—the original string is unchanged.

37. (a) 0

(b) A null pointer exception is thrown.

38. It is a reference of type String.

39.

```
public void addItems(int quantity, double price)
{
    for (int i = 1; i <= quantity; i++)
    {
        this.addItem(price);
    }
}
```

40. System.in and System.out

41. Math.PI

42. The method needs no data of any object. The only required input is the values argument.

43. Yes, it works. Static methods can call each other and access static variables—any method can. But it is a terrible idea. A program that consists of a single class with many methods is hard to understand.

44. It needs to be incremented in the deposit and withdraw methods. There also needs to be some method to reset it after the end of a statement period.

45. The ArrayList<String> instance variable is private, and the class users cannot access it.

46. Add an ArrayList<Double> prices. In the addItem method, add the current price. In the reset method, replace the array list with an empty one. Also supply a method printReceipt that prints the prices.

47. The tax ID of an employee does not change, and no setter method should be supplied. The salary of an employee can change, and both getter and setter methods should be supplied.

48. It is an example of the “state pattern” described in Section 8.11.5. The direction is a state that changes when the bug turns, and it affects how the bug moves.

49. (a) No; (b) Yes; (c) Yes; (d) No

50. No—you can use fully qualified names for all other classes, such as java.util.Random and java.awt.Rectangle.

51. /home/me/cs101/hw1/problem1 or, on Windows, c:\Users\Me\cs101\hw1\problem1.

WORKED EXAMPLE 8.1

Implementing a Menu Class



Problem Statement In this Worked Example, our task is to write a class for displaying a menu. An object of this class can display a menu such as

- 1) Open new account
- 2) Log into existing account
- 3) Help
- 4) Quit

Then the menu waits for the user to supply a value. If the user does not supply a valid value, the menu is redisplayed, and the user can try again.

Step 1 Get an informal list of the responsibilities of your objects.

The basic responsibilities of a menu are quite simple:

- Display the menu.
- Get user input.

Now consider how a menu is produced. The programmer creates an empty menu object and then adds options “Open new account”, “Help”, and so on. That is another responsibility:

- Add an option.

Step 2 Specify the public interface.

It is a good idea to write out method calls that are applied to a sample object, like this:

```
Menu mainMenu = new Menu();
mainMenu.addOption("Open new account");
// Add more options
int input = mainMenu.getInput();
```

Now we have a specific list of methods.

- void addOption(String option)
- int getInput()

Let's look at the sample code more closely. There is no sense in displaying the menu without also asking the user for input. It would be better if the `getInput` method displays the menu. In fact, `getInput` may need to display the menu more than once if the user provides a bad input. Thus, `display` is a good candidate for a private method.

To complete the public interface, you need to specify the constructors. In the case of the menu example, we can get by with a single constructor that creates an empty menu.

Here is the public interface:

```
public class Menu
{
    public Menu() { . . . }
    public void addOption(String option) { . . . }
    public int getInput() { . . . }
}
```

Step 3 Document the public interface.

Supply a documentation comment for the class, then comment each method.

```
/**
 * A menu that is displayed on a console.
 */
public class Menu
{
```

WE2 Chapter 8 Objects and Classes

```
/**  
 * Constructs a menu with no options.  
 */  
public Menu() { . . . }  
  
/**  
 * Adds an option to the end of this menu.  
 * @param option the option to add  
 */  
public void addOption(String option) { . . . }  
  
/**  
 * Displays the menu, with options numbered starting with 1,  
 * and prompts the user for input. Repeats until a valid input  
 * is supplied.  
 * @return the number that the user supplied  
 */  
public int getInput() { . . . }  
}
```

Step 4 Determine instance variables.

Let's start with the `addOption` method. We clearly need to store the added menu option so that the menu can be displayed later. How should we store the options? As an array list of strings? As one long string? Both approaches can be made to work. We will use an array list here. Exercise E8.5 asks you to implement the other approach.

```
public class Menu  
{  
    private ArrayList<String> options;  
    . . .  
}
```

Now consider the `getInput` method. It shows the stored options and reads an integer. When checking that the input is valid, we need to know the number of menu items. Because we store them in an array list, the number of menu items is simply obtained as the size of the array list. If you stored the menu items in one long string, you might want to keep another instance variable that stores the item count.

We will also need a scanner to read the user input, which we will add as another instance variable:

```
private Scanner in;
```

Step 5 Implement constructors and methods.

Let's start with the simplest method. Here is the implementation of the `addOption` method:

```
public void addOption(String option)  
{  
    options.add(option);  
}
```

Here is the `getInput` method. This method is a bit more sophisticated. It loops until a valid input has been obtained, displaying the menu options before reading the input.

```
public int getInput()  
{  
    int input;  
    do  
    {  
        for (int i = 0; i < options.size(); i++)  
        {
```

```

        int choice = i + 1;
        System.out.println(choice + " " + options.get(i));
    }
    input = in.nextInt();
}
while (input < 1 || input > options.size());
return input;
}

```

Finally, we need to supply a constructor to initialize the instance variables:

```

public Menu()
{
    options = new ArrayList<String>();
    in = new Scanner(System.in);
}

```

Step 6 Test your class.

Here is the tester program that carries out the method calls that you found in Step 2.

```

public class MenuTester
{
    public static void main(String[] args)
    {
        Menu mainMenu = new Menu();
        mainMenu.addOption("Open new account");
        mainMenu.addOption("Log into existing account");
        mainMenu.addOption("Help");
        mainMenu.addOption("Quit");
        int input = mainMenu.getInput();
        System.out.println("Input: " + input);
    }
}

```

Program Run

```

1) Open new account
2) Log into existing account
3) Help
4) Quit
5
1) Open new account
2) Log into existing account
3) Help
4) Quit
3
Input: 3

```

Here's the complete program:

worked_example_1/Menu.java

```

1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 /**
5  * A menu that is displayed on a console.
6 */
7 public class Menu
8 {

```

WE4 Chapter 8 Objects and Classes

```
 9  private ArrayList<String> options;
10 private Scanner in;
11
12 /**
13  * Constructs a menu with no options.
14 */
15 public Menu()
16 {
17     options = new ArrayList<String>();
18     in = new Scanner(System.in);
19 }
20
21 /**
22  * Adds an option to the end of this menu.
23  * @param option the option to add
24 */
25 public void addOption(String option)
26 {
27     options.add(option);
28 }
29
30 /**
31  * Displays the menu, with options numbered starting with 1,
32  * and prompts the user for input. Repeats until a valid input
33  * is supplied.
34  * @return the number that the user supplied
35 */
36 public int getInput()
37 {
38     int input;
39     do
40     {
41         for (int i = 0; i < options.size(); i++)
42         {
43             int choice = i + 1;
44             System.out.println(choice + " " + options.get(i));
45         }
46         input = in.nextInt();
47     }
48     while (input < 1 || input > options.size());
49     return input;
50 }
51 }
```

worked_example_1/MenuTester.java

```
1 /**
2  * This program tests the menu class.
3 */
4 public class MenuTester
5 {
6     public static void main(String[] args)
7     {
8         Menu mainMenu = new Menu();
9         mainMenu.addOption("Open new account");
10        mainMenu.addOption("Log into existing account");
```

```
11     mainMenu.addOption("Help");
12     mainMenu.addOption("Quit");
13     int input = mainMenu.getInput();
14     System.out.println("Input: " + input);
15 }
16 }
```


INHERITANCE AND INTERFACES

CHAPTER GOALS

- To learn about inheritance
- To implement subclasses that inherit and override superclass methods
- To understand the concept of polymorphism
- To be familiar with the common superclass `Object` and its methods
- To work with interface types

CHAPTER CONTENTS

9.1 INHERITANCE HIERARCHIES 438

PT1 Use a Single Class for Variation in Values, Inheritance for Variation in Behavior 442

9.2 IMPLEMENTING SUBCLASSES 442

SYN Subclass Declaration 444

CE1 Replicating Instance Variables from the Superclass 445

CE2 Confusing Super- and Subclasses 446

9.3 OVERRIDING METHODS 446

CE3 Accidental Overloading 450

CE4 Forgetting to Use `super` When Invoking a Superclass Method 451

ST1 Calling the Superclass Constructor 451

SYN Constructor with Superclass Initializer 452

9.4 POLYMORPHISM 452

ST2 Dynamic Method Lookup and the Implicit Parameter 455

ST3 Abstract Classes 456

ST4 Final Methods and Classes 457

ST5 Protected Access 458

HT1 Developing an Inheritance Hierarchy 458

WE1 Implementing an Employee Hierarchy for Payroll Processing 



© Lisa Thornberg/iStockphoto.

VE1 Building a Discussion Board 

9.5 OBJECT: THE COSMIC SUPERCLASS 463

SYN The `instanceof` Operator 467

CE5 Don't Use Type Tests 468

ST6 Inheritance and the `toString` Method 468

ST7 Inheritance and the `equals` Method 469

9.6 INTERFACE TYPES 470

SYN Interface Types 471

CE6 Forgetting to Declare Implementing Methods as Public 475

PT2 Comparing Integers and Floating-Point Numbers 475

ST8 Constants in Interfaces 476

ST9 Generic Interface Types 476

J81 Static Methods in Interfaces 477

J82 Default Methods 477

ST10 Function Objects 478

J83 Lambda Expressions 479

WE2 Investigating Number Sequences 

VE2 Drawing Geometric Shapes 

C&S Who Controls the Internet? 481



© Lisa Thorberg/iStockphoto.

Objects from related classes usually share common behavior. For example, shovels, rakes, and clippers all perform gardening tasks. In this chapter, you will learn how the notion of inheritance expresses the relationship between specialized and general classes. By using inheritance, you will be able to share code between classes and provide services that can be used by multiple classes.

9.1 Inheritance Hierarchies

A subclass inherits data and behavior from a superclass.

You can always use a subclass object in place of a superclass object.

In object-oriented design, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**). The subclass inherits data and behavior from the superclass. For example, consider the relationships between different kinds of vehicles depicted in Figure 1.

Every car *is a* vehicle. Cars share the common traits of all vehicles, such as the ability to transport people from one place to another. We say that the class `Car` inherits from the class `Vehicle`. In this relationship, the `Vehicle` class is the superclass and the `Car` class is the subclass. In Figure 2, the superclass and subclass are joined with an arrow that points to the superclass.

Suppose we have an algorithm that manipulates a `Vehicle` object. Because a car is a special kind of vehicle, we can use a `Car` object in such an algorithm, and it will work correctly. The **substitution principle** states that you can always use a subclass object when a superclass object is expected. For example, consider a method that takes an argument of type `Vehicle`:

```
void processVehicle(Vehicle v)
```

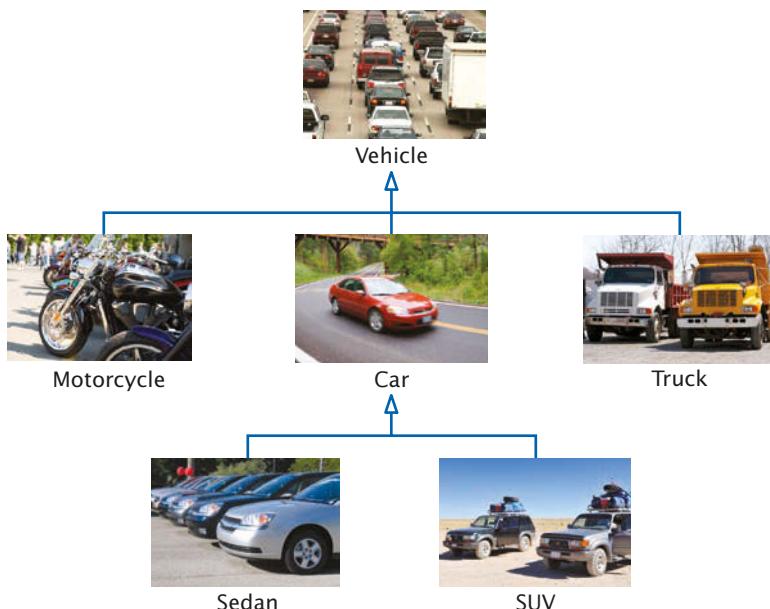
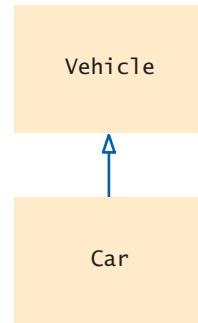


Figure 1 An Inheritance Hierarchy of Vehicle Classes

© Richard Stouffer/iStockphoto (vehicle); © Ed Hidden/iStockphoto (motorcycle);
© Yin Yang/iStockphoto (car); © Robert Pernell/iStockphoto (truck);
© nicholas belton/iStockphoto (sedan); © Cezary Wojtkowski/Age Fotostock America (SUV).

Figure 2
An Inheritance Diagram



Because `Car` is a subclass of `Vehicle`, you can call that method with a `Car` object:

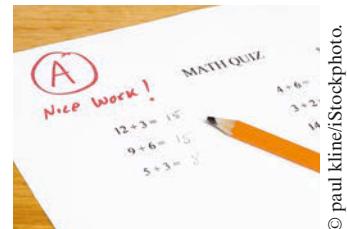
```
Car myCar = new Car(. . .);
processVehicle(myCar);
```

Why provide a method that processes `Vehicle` objects instead of `Car` objects? That method is more useful because it can handle *any* kind of vehicle (including `Truck` and `Motorcycle` objects). In general, when we group classes into an inheritance hierarchy, we can share common code among the classes.

In this chapter, we will consider a simple hierarchy of classes. Most likely, you have taken computer-graded quizzes. A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok; e.g., 1.33 when the actual answer is $4/3$)
- Free response

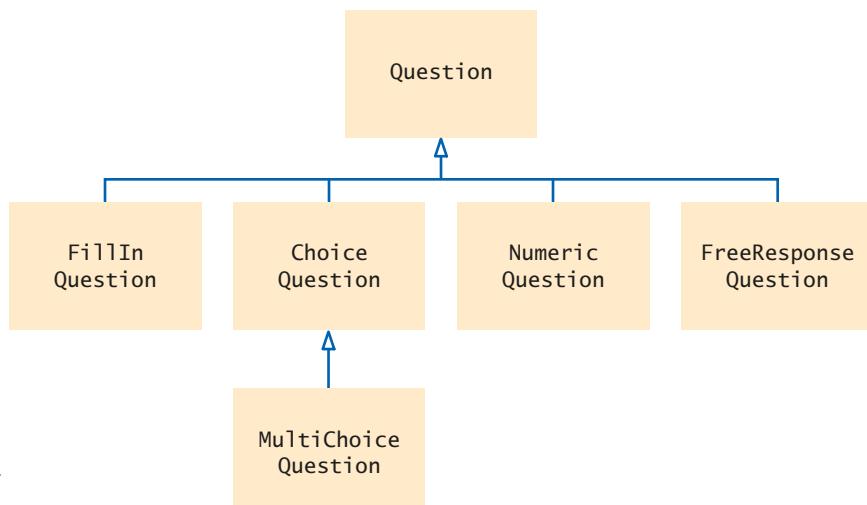
Figure 3 shows an inheritance hierarchy for these question types.



© paul kline/Stockphoto

We will develop a simple but flexible quiz-taking program to illustrate inheritance.

Figure 3
Inheritance Hierarchy
of Question Types



At the root of this hierarchy is the `Question` type. A question can display its text, and it can check whether a given response is a correct answer.

sec01/Question.java

```
1  /**
2   *      A question with a text and an answer.
3  */
4  public class Question
5  {
6      private String text;
7      private String answer;
8
9      /**
10       * Constructs a question with empty question and answer.
11      */
12     public Question()
13     {
14         text = "";
15         answer = "";
16     }
17
18     /**
19      * Sets the question text.
20      * @param questionText the text of this question
21     */
22     public void setText(String questionText)
23     {
24         text = questionText;
25     }
26
27     /**
28      * Sets the answer for this question.
29      * @param correctResponse the answer
30     */
31     public void setAnswer(String correctResponse)
32     {
33         answer = correctResponse;
34     }
35
36     /**
37      * Checks a given response for correctness.
38      * @param response the response to check
39      * @return true if the response was correct, false otherwise
40     */
41     public boolean checkAnswer(String response)
42     {
43         return response.equals(answer);
44     }
45
46     /**
47      * Displays this question.
48     */
49     public void display()
50     {
51         System.out.println(text);
52     }
53 }
```

This question class is very basic. It does not handle multiple-choice questions, numeric questions, and so on. In the following sections, you will see how to form subclasses of the `Question` class.

Here is a simple test program for the `Question` class:

sec01/QuestionDemo1.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program shows a simple quiz with one question.
5 */
6 public class QuestionDemo1
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11
12         Question q = new Question();
13         q.setText("Who was the inventor of Java?");
14         q.setAnswer("James Gosling");
15
16         q.display();
17         System.out.print("Your answer: ");
18         String response = in.nextLine();
19         System.out.println(q.checkAnswer(response));
20     }
21 }
```

Program Run

```

Who was the inventor of Java?
Your answer: James Gosling
true
```



1. Consider classes `Manager` and `Employee`. Which should be the superclass and which should be the subclass?
2. What are the inheritance relationships between classes `BankAccount`, `CheckingAccount`, and `SavingsAccount`?
3. Figure 7.2 shows an inheritance diagram of exception classes in Java. List all superclasses of the class `RuntimeException`.
4. Consider the method `doSomething(Car c)`. List all vehicle classes from Figure 1 whose objects *cannot* be passed to this method.
5. Should a class `Quiz` inherit from the class `Question`? Why or why not?

Practice It Now you can try these exercises at the end of the chapter: R9.1, R9.7, R9.9.

Programming Tip 9.1



Use a Single Class for Variation in Values, Inheritance for Variation in Behavior

The purpose of inheritance is to model objects with different *behavior*. When students first learn about inheritance, they have a tendency to overuse it, by creating multiple classes even though the variation could be expressed with a simple instance variable.

Consider a program that tracks the fuel efficiency of a fleet of cars by logging the distance traveled and the refueling amounts. Some cars in the fleet are hybrids. Should you create a subclass `HybridCar`? Not in this application. Hybrids don't behave any differently than other cars when it comes to driving and refueling. They just have a better fuel efficiency. A single `Car` class with an instance variable

```
double milesPerGallon;
```

is entirely sufficient.

However, if you write a program that shows how to repair different kinds of vehicles, then it makes sense to have a separate class `HybridCar`. When it comes to repairs, hybrid cars behave differently from other cars.

9.2 Implementing Subclasses

In this section, you will see how to form a subclass and how a subclass automatically inherits functionality from its superclass.

Suppose you want to write a program that handles questions such as the following:

In which country was the inventor of Java born?

1. Australia
2. Canada
3. Denmark
4. United States

You could write a `ChoiceQuestion` class from scratch, with methods to set up the question, display it, and check the answer. But you don't have to. Instead, use inheritance and implement `ChoiceQuestion` as a subclass of the `Question` class (see Figure 4).

In Java, you form a subclass by specifying what makes the subclass different from its superclass.

Subclass objects automatically have the instance variables that are declared in the superclass. You only declare instance variables that are not part of the superclass objects.

A subclass inherits all methods that it does not override.

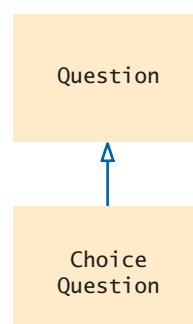


Figure 4
The `ChoiceQuestion` Class is a Subclass of the `Question` Class

Like the manufacturer of a stretch limo, who starts with a regular car and modifies it, a programmer makes a subclass by modifying another class.



Media Bakery.

A subclass can override a superclass method by providing a new implementation.

The subclass inherits all public methods from the superclass. You declare any methods that are *new* to the subclass, and *change* the implementation of inherited methods if the inherited behavior is not appropriate. When you supply a new implementation for an inherited method, you **override** the method.

A ChoiceQuestion object differs from a Question object in three ways:

- Its objects store the various choices for the answer.
- There is a method for adding answer choices.
- The `display` method of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

When the `ChoiceQuestion` class inherits from the `Question` class, it needs to spell out these three differences:

```
public class ChoiceQuestion extends Question
{
    // This instance variable is added to the subclass
    private ArrayList<String> choices;

    // This method is added to the subclass
    public void addChoice(String choice, boolean correct) { . . . }

    // This method overrides a method from the superclass
    public void display() { . . . }
}
```

The `extends` reserved word indicates that a class inherits from a superclass.

The reserved word `extends` denotes inheritance.

Figure 5 shows the layout of a `ChoiceQuestion` object. It has the `text` and `answer` instance variables that are declared in the `Question` superclass, and it adds an additional instance variable, `choices`.

The `addChoice` method is specific to the `ChoiceQuestion` class. You can only apply it to `ChoiceQuestion` objects, not general `Question` objects.

In contrast, the `display` method is a method that already exists in the superclass. The subclass overrides this method, so that the choices can be properly displayed.

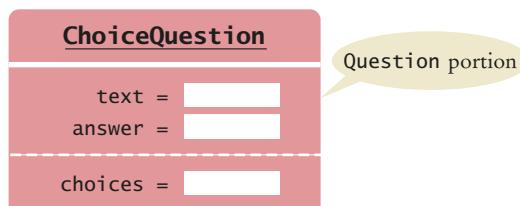


Figure 5 Data Layout of Subclass Object

Syntax 9.1 Subclass Declaration

```
Syntax public class SubclassName extends SuperclassName
{
    instance variables
    methods
}
```

The reserved word **extends** denotes inheritance.

Declare instance variables that are **added** to the subclass.

Declare methods that are **added** to the subclass.

Declare methods that the subclass **overrides**.

```
Subclass           Superclass
```

```
public class ChoiceQuestion extends Question
{
    private ArrayList<String> choices;
    public void addChoice(String choice, boolean correct) { . . . }

    public void display() { . . . }
}
```

All other methods of the Question class are automatically inherited by the ChoiceQuestion class.

You can call the inherited methods on a subclass object:

```
choiceQuestion.setAnswer("2");
```

However, the private instance variables of the superclass are inaccessible. Because these variables are private data of the superclass, only the superclass has access to them. The subclass has no more access rights than any other class.

In particular, the ChoiceQuestion methods cannot directly access the instance variable answer. These methods must use the public interface of the Question class to access its private data, just like every other method.

To illustrate this point, let's implement the addChoice method. The method has two arguments: the choice to be added (which is appended to the list of choices), and a Boolean value to indicate whether this choice is correct. For example,

```
question.addChoice("Canada", true);
```

The first argument is added to the choices variable. If the second argument is true, then the answer instance variable becomes the number of the current choice. For example, if choices.size() is 2, then answer is set to the string "2".

```
public void addChoice(String choice, boolean correct)
{
    choices.add(choice);
    if (correct)
    {
        // Convert choices.size() to string
        String choiceString = "" + choices.size();
        setAnswer(choiceString);
    }
}
```

You can't just access the answer variable in the superclass. Fortunately, the Question class has a setAnswer method. You can call that method. On which object? The

**FULL CODE EXAMPLE**

Go to wiley.com/go/bj102code to download a program that shows a simple Car class extending a Vehicle class.

SELF CHECK

6. Suppose q is an object of the class Question and cq an object of the class ChoiceQuestion. Which of the following calls are legal?

- a. q.setAnswer(response)
- b. cq.setAnswer(response)
- c. q.addChoice(choice, true)
- d. cq.addChoice(choice, true)

7. Suppose the class Employee is declared as follows:

```
public class Employee
{
    private String name;
    private double baseSalary;

    public void setName(String newName) { . . . }
    public void setBaseSalary(double newSalary) { . . . }
    public String getName() { . . . }
    public double getSalary() { . . . }
}
```

Declare a class Manager that inherits from the class Employee and adds an instance variable bonus for storing a salary bonus. Omit constructors and methods.

- 8. Which instance variables does the Manager class from Self Check 7 have?
- 9. In the Manager class, provide the method header (but not the implementation) for a method that overrides the getSalary method from the class Employee.
- 10. Which methods does the Manager class from Self Check 9 inherit?

Practice It

Now you can try these exercises at the end of the chapter: R9.3, E9.6, E9.13.

Common Error 9.1**Replicating Instance Variables from the Superclass**

A subclass has no access to the private instance variables of the superclass.

```
public ChoiceQuestion(String questionText)
{
    text = questionText; // Error—tries to access private superclass variable
}
```

When faced with a compiler error, beginners commonly “solve” this issue by adding *another* instance variable with the same name to the subclass:

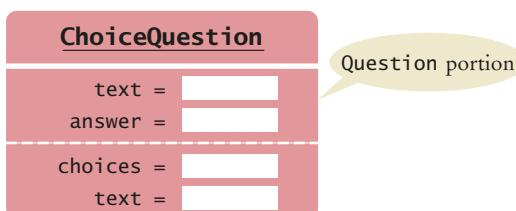
```
public class ChoiceQuestion extends Question
{
```

```

private ArrayList<String> choices;
private String text; // Don't!
...
}

```

Sure, now the constructor compiles, but it doesn't set the correct text! Such a `ChoiceQuestion` object has two instance variables, both named `text`. The constructor sets one of them, and the `display` method displays the other.



Common Error 9.2



Confusing Super- and Subclasses

If you compare an object of type `ChoiceQuestion` with an object of type `Question`, you find that

- The reserved word `extends` suggests that the `ChoiceQuestion` object is an extended version of a `Question`.
- The `ChoiceQuestion` object is larger; it has an added instance variable, `choices`.
- The `ChoiceQuestion` object is more capable; it has an `addChoice` method.

It seems a superior object in every way. So why is `ChoiceQuestion` called the *subclass* and `Question` the *superclass*?

The *super/sub* terminology comes from set theory. Look at the set of all questions. Not all of them are `ChoiceQuestion` objects; some of them are other kinds of questions. Therefore, the set of `ChoiceQuestion` objects is a *subset* of the set of all `Question` objects, and the set of `Question` objects is a *superset* of the set of `ChoiceQuestion` objects. The more specialized objects in the subset have a richer state and more capabilities.

9.3 Overriding Methods

An overriding method can extend or replace the functionality of the superclass method.

The subclass inherits the methods from the superclass. If you are not satisfied with the behavior of an inherited method, you *override* it by specifying a new implementation in the subclass.

Consider the `display` method of the `ChoiceQuestion` class. It overrides the superclass `display` method in order to show the choices for the answer. This method *extends* the functionality of the superclass version. This means that the subclass method carries out the action of the superclass method (in our case, displaying the question text), and it also does some additional work (in our case, displaying the choices). In other cases, a subclass method *replaces* the functionality of a superclass method, implementing an entirely different behavior.

Let us turn to the implementation of the `display` method of the `ChoiceQuestion` class. The method needs to

- Display the question text.
- Display the answer choices.

The second part is easy because the answer choices are an instance variable of the subclass.

```
public class ChoiceQuestion
{
    . .
    public void display()
    {
        // Display the question text
        . .
        // Display the answer choices
        for (int i = 0; i < choices.size(); i++)
        {
            int choiceNumber = i + 1;
            System.out.println(choiceNumber + ": " + choices.get(i));
        }
    }
}
```

But how do you get the question text? You can't access the text variable of the superclass directly because it is private.

Instead, you can call the `display` method of the superclass, by using the reserved word `super`:

```
public void display()
{
    // Display the question text
    super.display(); // OK
    // Display the answer choices
    . .
}
```

If you omit the reserved word `super`, then the method will not work as intended.

```
public void display()
{
    // Display the question text
    display(); // Error—invokes this.display()
    . .
}
```

Because the implicit parameter `this` is of type `ChoiceQuestion`, and there is a method named `display` in the `ChoiceQuestion` class, that method will be called—but that is just the method you are currently writing! The method would call itself over and over.

Here is the complete program that lets you take a quiz consisting of two `ChoiceQuestion` objects. We construct both objects and pass them to a method `presentQuestion`. That method displays the question to the user and checks whether the user response is correct.

sec03/QuestionDemo2.java

```
1 import java.util.Scanner;
2 /**
3  * This program shows a simple quiz with two choice questions.
4  */
5 public class QuestionDemo2
6 {
7     public static void main(String[] args)
8     {
9 }
```

```

10    ChoiceQuestion first = new ChoiceQuestion();
11    first.setText("What was the original name of the Java language?");
12    first.addChoice("7", false);
13    first.addChoice("Duke", false);
14    first.addChoice("Oak", true);
15    first.addChoice("Gosling", false);
16
17    ChoiceQuestion second = new ChoiceQuestion();
18    second.setText("In which country was the inventor of Java born?");
19    second.addChoice("Australia", false);
20    second.addChoice("Canada", true);
21    second.addChoice("Denmark", false);
22    second.addChoice("United States", false);
23
24    presentQuestion(first);
25    presentQuestion(second);
26 }
27
28 /**
29  * Presents a question to the user and checks the response.
30  * @param q the question
31 */
32 public static void presentQuestion(ChoiceQuestion q)
33 {
34     q.display();
35     System.out.print("Your answer: ");
36     Scanner in = new Scanner(System.in);
37     String response = in.nextLine();
38     System.out.println(q.checkAnswer(response));
39 }
40 }
```

sec03/ChoiceQuestion.java

```

1 import java.util.ArrayList;
2
3 /**
4  * A question with multiple choices.
5 */
6 public class ChoiceQuestion extends Question
7 {
8     private ArrayList<String> choices;
9
10    /**
11     * Constructs a choice question with no choices.
12     */
13    public ChoiceQuestion()
14    {
15        choices = new ArrayList<String>();
16    }
17
18    /**
19     * Adds an answer choice to this question.
20     * @param choice the choice to add
21     * @param correct true if this is the correct choice, false otherwise
22     */
23    public void addChoice(String choice, boolean correct)
24    {
```

```

25     choices.add(choice);
26     if (correct)
27     {
28         // Convert choices.size() to string
29         String choiceString = "" + choices.size();
30         setAnswer(choiceString);
31     }
32 }
33
34 public void display()
35 {
36     // Display the question text
37     super.display();
38     // Display the answer choices
39     for (int i = 0; i < choices.size(); i++)
40     {
41         int choiceNumber = i + 1;
42         System.out.println(choiceNumber + ": " + choices.get(i));
43     }
44 }
45 }
```

Program Run

What was the original name of the Java language?

1: *7
 2: Duke
 3: Oak
 4: Gosling
 Your answer: *7
 false

In which country was the inventor of Java born?

1: Australia
 2: Canada
 3: Denmark
 4: United States
 Your answer: 2
 true

SELF CHECK



11. What is wrong with the following implementation of the `display` method?

```

public class ChoiceQuestion
{
    .
    .
    public void display()
    {
        System.out.println(text);
        for (int i = 0; i < choices.size(); i++)
        {
            int choiceNumber = i + 1;
            System.out.println(choiceNumber + ": " + choices.get(i));
        }
    }
}
```

12. What is wrong with the following implementation of the `display` method?

```

public class ChoiceQuestion
{
```

```

        .
        .
        public void display()
{
    this.display();
    for (int i = 0; i < choices.size(); i++)
    {
        int choiceNumber = i + 1;
        System.out.println(choiceNumber + ": " + choices.get(i));
    }
}
}

```

13. Look again at the implementation of the `addChoice` method that calls the `setAnswer` method of the superclass. Why don't you need to call `super.setAnswer`?
14. In the `Manager` class of Self Check 7, override the `getName` method so that managers have a * before their name (such as *Lin, Sally).
15. In the `Manager` class of Self Check 9, override the `getSalary` method so that it returns the sum of the salary and the bonus.

Practice It Now you can try these exercises at the end of the chapter: E9.1, E9.2, E9.14.

Common Error 9.3



Accidental Overloading

In Java, two methods can have the same name, provided they differ in their parameter types. For example, the `PrintStream` class has methods called `println` with headers

`void println(int x)`
and

`void println(String x)`

These are different methods, each with its own implementation. The Java compiler considers them to be completely unrelated. We say that the `println` name is **overloaded**. This is different from overriding, where a subclass method provides an implementation of a method whose parameter variables have the *same* types.

If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method. For example,

```

public class ChoiceQuestion extends Question
{
    .
    .
    public void display(PrintStream out)
    // Does not override void display()
    {
        .
        .
    }
}

```

The compiler will not complain. It thinks that you want to provide a method just for `PrintStream` arguments, while inheriting another method `void display()`.

When overriding a method, be sure to check that the types of the parameter variables match exactly.

Common Error 9.4**Forgetting to Use super When Invoking a Superclass Method**

A common error in extending the functionality of a superclass method is to forget the reserved word `super`. For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
public class Manager
{
    ...
    public double getSalary()
    {
        double baseSalary = getSalary();
        // Error: should be super.getSalary()
        return baseSalary + bonus;
    }
}
```

Here `getSalary()` refers to the `getSalary` method applied to the implicit parameter of the method. The implicit parameter is of type `Manager`, and there is a `getSalary` method in the `Manager` class. Calling that method is a recursive call, which will never stop. Instead, you must tell the compiler to invoke the superclass method.

Whenever you call a superclass method from a subclass method with the same name, be sure to use the reserved word `super`.

Special Topic 9.1**Calling the Superclass Constructor**

Consider the process of constructing a subclass object. A subclass constructor can only initialize the instance variables of the subclass. But the superclass instance variables also need to be initialized. Unless you specify otherwise, the superclass instance variables are initialized with the constructor of the superclass that has no arguments.

In order to specify another constructor, you use the `super` reserved word, together with the arguments of the superclass constructor, as the *first statement* of the subclass constructor.

For example, suppose the `Question` superclass had a constructor for setting the question text. Here is how a subclass constructor could call that superclass constructor:

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>();
}
```

In our example program, we used the superclass constructor with no arguments. However, if all superclass constructors have arguments, you must use the `super` syntax and provide the arguments for a superclass constructor.

When the reserved word `super` is followed by a parenthesis, it indicates a call to the superclass constructor. When used in this way, the constructor call must be the *first statement of the subclass constructor*. If `super` is followed by a period and a method name, on the other hand, it indicates a call to a superclass method, as you saw in the preceding section. Such a call can be made anywhere in any subclass method.

Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.

To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.

The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Syntax 9.2 Constructor with Superclass Initializer

```
Syntax     public ClassName(parameterType parameterName, . . .)
            {
                super(arguments);
            }
```

The superclass constructor is called first.

The constructor body can contain additional statements.

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>;
```

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

9.4 Polymorphism

In this section, you will learn how to use inheritance for processing objects of different types in the same program.

Consider our first sample program. It presented two `Question` objects to the user. The second sample program presented two `ChoiceQuestion` objects. Can we write a program that shows a mixture of both question types?

With inheritance, this goal is very easy to realize. In order to present a question to the user, we need not know the exact type of the question. We just display the question and check whether the user supplied the correct answer. The `Question` superclass has methods for this purpose. Therefore, we can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`:

```
public static void presentQuestion(Question q)
{
    q.display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));
}
```

As discussed in Section 9.1, we can substitute a subclass object whenever a superclass object is expected:

```
ChoiceQuestion second = new ChoiceQuestion();
.
.
.
presentQuestion(second); // OK to pass a ChoiceQuestion
```

A subclass reference can be used when a superclass reference is expected.

When the `presentQuestion` method executes, the object references stored in `second` and `q` refer to the same object of type `ChoiceQuestion` (see Figure 6).

However, the *variable* `q` knows less than the full story about the object to which it refers (see Figure 7).

Because `q` is a variable of type `Question`, you can call the `display` and `checkAnswer` methods. You cannot call the `addChoice` method, though—it is not a method of the `Question` superclass.

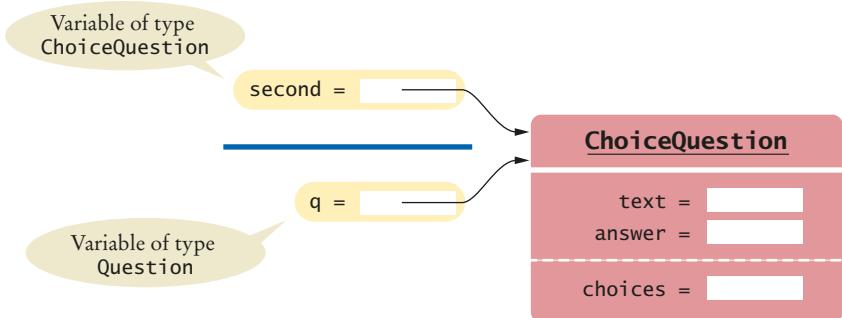


Figure 6 Variables of Different Types Referring to the Same Object

This is as it should be. After all, it happens that in this method call, `q` refers to a `ChoiceQuestion`. In another method call, `q` might refer to a plain `Question` or an entirely different subclass of `Question`.

Now let's have a closer look inside the `presentQuestion` method. It starts with the call

```
q.display(); // Does it call Question.display or ChoiceQuestion.display?
```

Which `display` method is called? If you look at the program output on page 455, you will see that the method called depends on the contents of the parameter variable `q`. In the first case, `q` refers to a `Question` object, so the `Question.display` method is called. But in the second case, `q` refers to a `ChoiceQuestion`, so the `ChoiceQuestion.display` method is called, showing the list of choices.

In Java, method calls *are always determined by the type of the actual object*, not the type of the variable containing the object reference. This is called **dynamic method lookup**.

Dynamic method lookup allows us to treat objects of different classes in a uniform way. This feature is called **polymorphism**. We ask multiple objects to carry out a task, and each object does so in its own way.

Polymorphism makes programs *easily extensible*. Suppose we want to have a new kind of question for calculations, where we are willing to accept an approximate answer. All we need to do is to declare a new class `NumericQuestion` that extends `Question`, with its own `checkAnswer` method. Then we can call the `presentQuestion` method with a mixture of plain questions, choice questions, and numeric questions. The `presentQuestion` method need not be changed at all! Thanks to dynamic method lookup, method calls to the `display` and `checkAnswer` methods automatically select the correct method of the newly declared classes.

Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

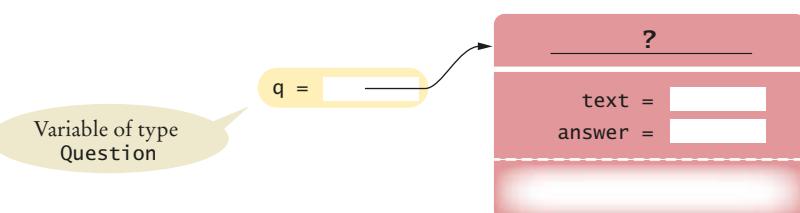


Figure 7 A Question Reference Can Refer to an Object of Any Subclass of Question



© Alipphoto/iStockphoto.

In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

sec04/QuestionDemo3.java

```

1 import java.util.Scanner;
2
3 /**
4     This program shows a simple quiz with two question types.
5 */
6 public class QuestionDemo3
7 {
8     public static void main(String[] args)
9     {
10         Question first = new Question();
11         first.setText("Who was the inventor of Java?");
12         first.setAnswer("James Gosling");
13
14         ChoiceQuestion second = new ChoiceQuestion();
15         second.setText("In which country was the inventor of Java born?");
16         second.addChoice("Australia", false);
17         second.addChoice("Canada", true);
18         second.addChoice("Denmark", false);
19         second.addChoice("United States", false);
20
21         presentQuestion(first);
22         presentQuestion(second);
23     }
24
25 /**
26     Presents a question to the user and checks the response.
27     @param q the question
28 */
29 public static void presentQuestion(Question q)
30 {
31     q.display();
32     System.out.print("Your answer: ");
33     Scanner in = new Scanner(System.in);
34     String response = in.nextLine();
35     System.out.println(q.checkAnswer(response));
36 }
37 }
```

Program Run

```

Who was the inventor of Java?
Your answer: Bjarne Stroustrup
false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2
true

```



SELF CHECK

16. Assuming `SavingsAccount` is a subclass of `BankAccount`, which of the following code fragments are valid in Java?
 - a. `BankAccount account = new SavingsAccount();`
 - b. `SavingsAccount account2 = new BankAccount();`
 - c. `BankAccount account = null;`
 - d. `SavingsAccount account2 = account;`
17. If `account` is a variable of type `BankAccount` that holds a non-null reference, what do you know about the object to which `account` refers?
18. Declare an array `quiz` that can hold a mixture of `Question` and `ChoiceQuestion` objects.
19. Consider the code fragment


```
ChoiceQuestion cq = . . .; // A non-null value
cq.display();
```

 Which actual method is being called?
20. Is the method call `Math.sqrt(2)` resolved through dynamic method lookup?

Practice It Now you can try these exercises at the end of the chapter: R9.6, E9.4, P9.5.

Special Topic 9.2



Dynamic Method Lookup and the Implicit Parameter

Suppose we add the `presentQuestion` method to the `Question` class itself:

```

public void presentQuestion()
{
    display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(checkAnswer(response));
}

```

Now consider the call

```

ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. .
cq.presentQuestion();

```

Which `display` and `checkAnswer` method will the `presentQuestion` method call? If you look inside the code of the `presentQuestion` method, you can see that these methods are executed on the implicit parameter.

```
public class Question
{
    public void presentQuestion()
    {
        this.display();
        System.out.print("Your answer: ");
        Scanner in = new Scanner(System.in);
        String response = in.nextLine();
        System.out.println(this.checkAnswer(response));
    }
}
```

The implicit parameter `this` in our call is a reference to an object of type `ChoiceQuestion`. Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display` and `checkAnswer` methods are called automatically. This happens even though the `presentQuestion` method is declared in the `Question` class, which has *no knowledge* of the `ChoiceQuestion` class.

As you can see, polymorphism is a very powerful mechanism. The `Question` class supplies a `presentQuestion` method that specifies the common nature of presenting a question, namely to display it and check the response. How the displaying and checking are carried out is left to the subclasses.

Special Topic 9.3



Abstract Classes

When you extend an existing class, you have the choice whether or not to override the methods of the superclass. Sometimes, it is desirable to *force* programmers to override a method. That happens when there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly.

Here is an example: Suppose the First National Bank of Java decides that every account type must have some monthly fees. Therefore, a `deductFees` method should be added to the `Account` class:

```
public class Account
{
    public void deductFees() { . . . }
    . .
}
```

But what should this method do? Of course, we could have the method do nothing. But then a programmer implementing a new subclass might simply forget to implement the `deductFees` method, and the new account would inherit the do-nothing method of the superclass. There is a better way—declare the `deductFees` method as an **abstract method**:

```
public abstract void deductFees();
```

An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. (Of course, some subclasses might decide to implement a do-nothing method, but then that is their choice—not a silently inherited default.)

You cannot construct objects of classes with abstract methods. For example, once the `Account` class has an abstract method, the compiler will flag an attempt to create a new `Account()` as an error.

An abstract method is a method whose implementation is not specified.

An abstract class is a class that cannot be instantiated.

A class for which you cannot create objects is called an **abstract class**. A class for which you can create objects is sometimes called a **concrete class**. In Java, you must declare all abstract classes with the reserved word `abstract`:

```
public abstract class Account
{
    public abstract void deductFees();
    . .
}

public class SavingsAccount extends Account // Not abstract
{
    public void deductFees() // Provides an implementation
    {
        . .
    }
}
```

Note that you cannot construct an *object* of an abstract class, but you can still have an *object reference* whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass:

```
Account anAccount; // OK
anAccount = new Account(); // Error—Account is abstract
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

The reason for using abstract classes is to force programmers to create subclasses. By specifying certain methods as abstract, you avoid the trouble of coming up with useless default methods that others might inherit by accident.

Special Topic 9.4



Final Methods and Classes

In Special Topic 9.3 you saw how you can force other programmers to create subclasses of abstract classes and override abstract methods. Occasionally, you may want to do the opposite and *prevent* other programmers from creating subclasses or from overriding certain methods. In these situations, you use the `final` reserved word. For example, the `String` class in the standard Java library has been declared as

```
public final class String { . . . }
```

That means that nobody can extend the `String` class. When you have a reference of type `String`, it must contain a `String` object, never an object of a subclass.

You can also declare individual methods as `final`:

```
public class SecureAccount extends BankAccount
{
    . .
    public final boolean checkPassword(String password)
    {
        . .
    }
}
```

This way, nobody can override the `checkPassword` method with another method that simply returns `true`.

Special Topic 9.5

**Protected Access**

We ran into a hurdle when trying to implement the `display` method of the `ChoiceQuestion` class. That method wanted to access the instance variable `text` of the superclass. Our remedy was to use the appropriate method of the superclass to display the text.

Java offers another solution to this problem. The superclass can declare an instance variable as *protected*:

```
public class Question
{
    protected String text;
    ...
}
```

Protected data in an object can be accessed by the methods of the object's class and all its subclasses. For example, `ChoiceQuestion` inherits from `Question`, so its methods can access the protected instance variables of the `Question` superclass.

Some programmers like the protected access feature because it seems to strike a balance between absolute protection (making instance variables private) and no protection at all (making instance variables public). However, experience has shown that protected instance variables are subject to the same kinds of problems as public instance variables. The designer of the superclass has no control over the authors of subclasses. Any of the subclass methods can corrupt the superclass data. Furthermore, classes with protected variables are hard to modify. Even if the author of the superclass would like to change the data implementation, the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them.

In Java, protected variables have another drawback—they are accessible not just by subclasses, but also by other classes in the same package.

It is best to leave all data private. If you want to grant access to the data to subclass methods only, consider making the *accessor* method protected.

HOW TO 9.1

**Developing an Inheritance Hierarchy**

When you work with a set of classes, some of which are more general and others more specialized, you want to organize them into an inheritance hierarchy. This enables you to process objects of different classes in a uniform way.

As an example, we will consider a bank that offers customers the following account types:

- A savings account that earns interest. The interest compounds monthly and is computed on the minimum monthly balance.
- A checking account that has no interest, gives you three free withdrawals per month, and charges a \$1 transaction fee for each additional withdrawal.

Problem Statement Design and implement a program that will manage a set of accounts of both types. It should be structured so that other account types can be added without affecting the main processing loop. Supply a menu

D)eposit W)ithdraw M)onth end Q)uit

For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.

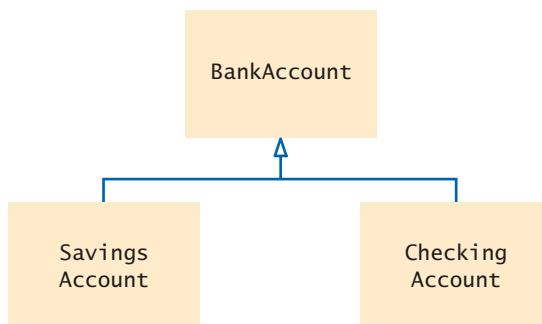
In the “Month end” command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description yields two classes: `SavingsAccount` and `CheckingAccount`. Of course, you could implement each of them separately. But that would not be a good idea because the classes would have to repeat common functionality, such as updating an account balance. We need another class that can be responsible for that common functionality. The problem statement does not explicitly mention such a class. Therefore, we need to discover it. Of course, in this case, the solution is simple. Savings accounts and checking accounts are special cases of a bank account. Therefore, we will introduce a common superclass `BankAccount`.

Step 2 Organize the classes into an inheritance hierarchy.

Draw an inheritance diagram that shows super- and subclasses. Here is one for our example:

**Step 3** Determine the common responsibilities.

In Step 2, you will have identified a class at the base of the hierarchy. That class needs to have sufficient responsibilities to carry out the tasks at hand. To find out what those tasks are, write pseudocode for processing the objects.

```

For each user command
  If it is a deposit or withdrawal
    Deposit or withdraw the amount from the specified account.
    Print the balance.
  If it is month end processing
    For each account
      Call month end processing.
      Print the balance.
  
```

From the pseudocode, we obtain the following list of common responsibilities that every bank account must carry out:

```

Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
  
```

Step 4 Decide which methods are overridden in subclasses.

For each subclass and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden. Be sure to declare any methods that are inherited or overridden in the root of the hierarchy.

```

public class BankAccount
{
  ...
  
```

```

    /**
     * Makes a deposit into this account.
     * @param amount the amount of the deposit
    */
    public void deposit(double amount) { . . . }

    /**
     * Makes a withdrawal from this account, or charges a penalty if
     * sufficient funds are not available.
     * @param amount the amount of the withdrawal
    */
    public void withdraw(double amount) { . . . }

    /**
     * Carries out the end of month processing that is appropriate
     * for this account.
    */
    public void monthEnd() { . . . }

    /**
     * Gets the current balance of this bank account.
     * @return the current balance
    */
    public double getBalance() { . . . }
}

```

The `SavingsAccount` and `CheckingAccount` classes both override the `monthEnd` method. The `SavingsAccount` class must also override the `withdraw` method to track the minimum balance. The `CheckingAccount` class must update a transaction count in the `withdraw` method.

Step 5 Declare the public interface of each subclass.

Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden. You also need to specify how the objects of the subclasses should be constructed.

In this example, we need a way of setting the interest rate for the savings account. In addition, we need to specify constructors and overridden methods.

```

public class SavingsAccount extends BankAccount
{
    . . .
    /**
     * Constructs a savings account with a zero balance.
    */
    public SavingsAccount() { . . . }

    /**
     * Sets the interest rate for this account.
     * @param rate the monthly interest rate in percent
    */
    public void setInterestRate(double rate) { . . . }

    // These methods override superclass methods
    public void withdraw(double amount) { . . . }
    public void monthEnd() { . . . }
}

public class CheckingAccount extends BankAccount
{
    . . .
    /**

```

```
Constructs a checking account with a zero balance.  
*/  
public CheckingAccount() { . . . }  
  
// These methods override superclass methods  
public void withdraw(double amount) { . . . }  
public void monthEnd() { . . . }  
}
```

Step 6 Identify instance variables.

List the instance variables for each class. If you find an instance variable that is common to all classes, be sure to place it in the base of the hierarchy.

All accounts have a balance. We store that value in the BankAccount superclass:

```
public class BankAccount  
{  
    private double balance;  
    . . .  
}
```

The SavingsAccount class needs to store the interest rate. It also needs to store the minimum monthly balance, which must be updated by all withdrawals.

```
public class SavingsAccount extends BankAccount  
{  
    private double interestRate;  
    private double minBalance;  
    . . .  
}
```

The CheckingAccount class needs to count the withdrawals, so that the charge can be applied after the free withdrawal limit is reached.

```
public class CheckingAccount extends BankAccount  
{  
    private int withdrawals;  
    . . .  
}
```

Step 7 Implement constructors and methods.

The methods of the BankAccount class update or return the balance.

```
public void deposit(double amount)  
{  
    balance = balance + amount;  
}  
  
public void withdraw(double amount)  
{  
    balance = balance - amount;  
}  
  
public double getBalance()  
{  
    return balance;  
}
```

At the level of the BankAccount superclass, we can say nothing about end of month processing. We choose to make that method do nothing:

```
public void monthEnd()  
{  
}
```

In the `withdraw` method of the `SavingsAccount` class, the minimum balance is updated. Note the call to the superclass method:

```
public void withdraw(double amount)
{
    super.withdraw(amount);
    double balance = getBalance();
    if (balance < minBalance)
    {
        minBalance = balance;
    }
}
```

In the `monthEnd` method of the `SavingsAccount` class, the interest is deposited into the account. We must call the `deposit` method because we have no direct access to the `balance` instance variable. The minimum balance is reset for the next month.

```
public void monthEnd()
{
    double interest = minBalance * interestRate / 100;
    deposit(interest);
    minBalance = getBalance();
}
```

The `withdraw` method of the `CheckingAccount` class needs to check the withdrawal count. If there have been too many withdrawals, a charge is applied. Again, note how the method invokes the superclass method:

```
public void withdraw(double amount)
{
    final int FREE_WITHDRAWALS = 3;
    final int WITHDRAWAL_FEE = 1;

    super.withdraw(amount);
    withdrawals++;
    if (withdrawals > FREE_WITHDRAWALS)
    {
        super.withdraw(WITHDRAWAL_FEE);
    }
}
```

End of month processing for a checking account simply resets the withdrawal count.

```
public void monthEnd()
{
    withdrawals = 0;
}
```

Step 8 Construct objects of different subclasses and process them.

In our sample program, we allocate 5 checking accounts and 5 savings accounts and store their addresses in an array of bank accounts. Then we accept user commands and execute deposits, withdrawals, and monthly processing.

```
BankAccount[] accounts = . . .;
. . .
Scanner in = new Scanner(System.in);
boolean done = false;
while (!done)
{
    System.out.print("D)deposit W)ithdraw M)onth end Q)uit: ");
    String input = in.next();
    if (input.equals("D") || input.equals("W")) // Deposit or withdrawal
    {
```

```

System.out.print("Enter account number and amount: ");
int num = in.nextInt();
double amount = in.nextDouble();

if (input.equals("D")) { accounts[num].deposit(amount); }
else { accounts[num].withdraw(amount); }

System.out.println("Balance: " + accounts[num].getBalance());
}
else if (input.equals("M")) // Month end processing
{
    for (int n = 0; n < accounts.length; n++)
    {
        accounts[n].monthEnd();
        System.out.println(n + " " + accounts[n].getBalance());
    }
}
else if (input.equals("Q"))
{
    done = true;
}
}
}

```

FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download a complete program with BankAccount, SavingsAccount, and CheckingAccount classes.

WORKED EXAMPLE 9.1**Implementing an Employee Hierarchy for Payroll Processing**

This Worked Example shows how to implement payroll processing that works for different kinds of employees. Go to wiley.com/go/bjlo2examples and download Worked Example 9.1.



© Jose Luis Pelaez Inc./
Getty Images Inc.

VIDEO EXAMPLE 9.1**Building a Discussion Board**

In this Video Example, you will learn how to build a discussion board for students and instructors. Go to wiley.com/go/bjlo2videos to view Video Example 9.1.



© vmi/Stockphoto.

9.5 Object: The Cosmic Superclass

In Java, every class that is declared without an explicit `extends` clause automatically extends the class `Object`. That is, the class `Object` is the direct or indirect superclass of *every* class in Java (see Figure 8). The `Object` class defines several very general methods, including

- `toString`, which yields a string describing the object (Section 9.5.1).
- `equals`, which compares objects with each other (Section 9.5.2).
- `hashCode`, which yields a numerical code for storing the object in a set (see Special Topic 15.1).

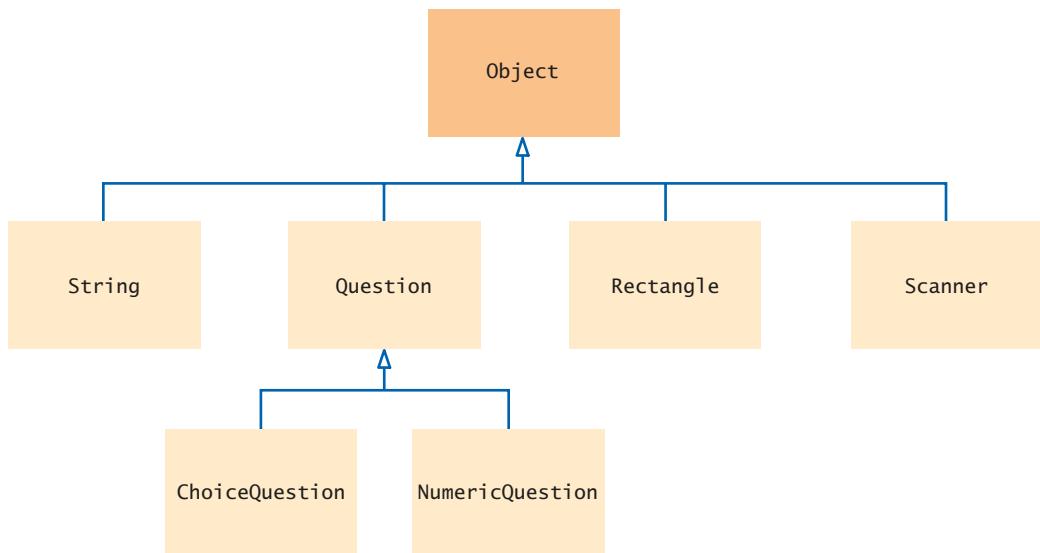


Figure 8 The `Object` Class Is the Superclass of Every Java Class

9.5.1 Overriding the `toString` Method

The `toString` method returns a string representation for each object. It is often used for debugging. For example, consider the `Rectangle` class in the standard Java library. Its `toString` method shows the state of a rectangle:

```

Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
  
```

The `toString` method is called automatically whenever you concatenate a string with an object. Here is an example:

```
"box=" + box;
```

On one side of the `+` concatenation operator is a string, but on the other side is an object reference. The Java compiler automatically invokes the `toString` method to turn the object into a string. Then both strings are concatenated. In this case, the result is the string

```
"box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

The compiler can invoke the `toString` method, because it knows that *every* object has a `toString` method: Every class extends the `Object` class, and that class declares `toString`.

As you know, numbers are also converted to strings when they are concatenated with other strings. For example,

```

int age = 18;
String s = "Harry's age is " + age;
// Sets s to "Harry's age is 18"
  
```

In this case, the `toString` method is *not* involved. Numbers are not objects, and there is no `toString` method for them. Fortunately, there is only a small set of primitive types, and the compiler knows how to convert them to strings.

Let's try the `toString` method for the `BankAccount` class:

```
BankAccount momSavings = new BankAccount(5000);
String s = momSavings.toString(); // Sets s to something like "BankAccount@d24606bf"
```

That's disappointing—all that's printed is the name of the class, followed by the **hash code**, a seemingly random code. The hash code can be used to tell objects apart—different objects are likely to have different hash codes. (See Special Topic 15.1 for the details.)

Override the `toString` method to yield a string that describes the object's state.

We don't care about the hash code. We want to know what is *inside* the object. But, of course, the `toString` method of the `Object` class does not know what is inside the `BankAccount` class. Therefore, we have to override the method and supply our own version in the `BankAccount` class. We'll follow the same format that the `toString` method of the `Rectangle` class uses: first print the name of the class, and then the values of the instance variables inside brackets.

```
public class BankAccount
{
    .
    .
    public String toString()
    {
        return "BankAccount[balance=" + balance + "]";
    }
}
```

This works better:

```
BankAccount momSavings = new BankAccount(5000);
String s = momSavings.toString(); // Sets s to "BankAccount[balance=5000]"
```

9.5.2 The `equals` Method

The `equals` method checks whether two objects have the same contents.

In addition to the `toString` method, the `Object` class also provides an `equals` method, whose purpose is to check whether two objects have the same contents:

```
if (stamp1.equals(stamp2)) . . . // Contents are the same—see Figure 9
```

This is different from the test with the `==` operator, which tests whether two references are identical, referring to the *same object*:

```
if (stamp1 == stamp2) . . . // Objects are the same—see Figure 10
```

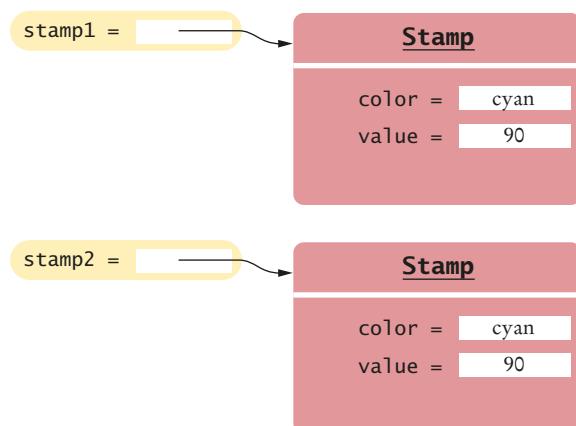


Figure 9 Two References to Equal Objects

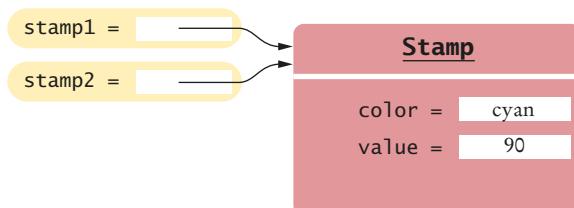


Figure 10 Two References to the Same Object

Let's implement the `equals` method for a `Stamp` class. You need to override the `equals` method of the `Object` class:

```
public class Stamp
{
    private String color;
    private int value;
    ...
    public boolean equals(Object otherObject)
    {
        ...
    }
    ...
}
```

Now you have a slight problem. The `Object` class knows nothing about stamps, so it declares the `otherObject` parameter variable of the `equals` method to have the type `Object`. When overriding the method, you are not allowed to change the type of the parameter variable. Cast the parameter variable to the class `Stamp`:

```
Stamp other = (Stamp) otherObject;
```

Then you can compare the two stamps:

```
public boolean equals(Object otherObject)
{
    Stamp other = (Stamp) otherObject;
    return color.equals(other.color)
        && value == other.value;
}
```

Note that this `equals` method can access the instance variables of *any* `Stamp` object: the access `other.color` is perfectly legal.



© Ken Brown/Stockphoto.

The `equals` method checks whether two objects have the same contents.

If you know that an object belongs to a given class, use a cast to convert the type.

The `instanceof` operator tests whether an object belongs to a particular type.

9.5.3 The `instanceof` Operator

As you have seen, it is legal to store a subclass reference in a superclass variable:

```
ChoiceQuestion cq = new ChoiceQuestion();
Question q = cq; // OK
Object obj = cq; // OK
```

Very occasionally, you need to carry out the opposite conversion, from a superclass reference to a subclass reference.

For example, you may have a variable of type `Object`, and you happen to know that it actually holds a `Question` reference. In that case, you can use a cast to convert the type:

```
Question q = (Question) obj;
```

However, this cast is somewhat dangerous. If you are wrong, and `obj` actually refers to an object of an unrelated type, then a “class cast” exception is thrown.

To protect against bad casts, you can use the `instanceof` operator. It tests whether an object belongs to a particular type. For example,

```
obj instanceof Question
```

returns true if the type of `obj` is convertible to `Question`. This happens if `obj` refers to an actual `Question` or to a subclass such as `ChoiceQuestion`.

Syntax 9.3 The instanceof Operator

Syntax *object instanceof TypeName*

If *anObject* is null,
instanceof returns false.

Returns true if *anObject*
can be cast to a *Question*.

```
if (anObject instanceof Question)
{
    Question q = (Question) anObject;
    ...
}
```

The object may belong to a
subclass of *Question*.

You can invoke *Question*
methods on this variable.

Two references
to the same object.

Using the instanceof operator, a safe cast can be programmed as follows:

```
if (obj instanceof Question)
{
    Question q = (Question) obj;
}
```

Note that instanceof is *not* a method. It is an operator, just like + or <. However, it does not operate on numbers. To the left is an object, and to the right a type name.

Do *not* use the instanceof operator to bypass polymorphism:

```
if (q instanceof ChoiceQuestion) // Don't do this—see Common Error 9.5
{
    // Do the task the ChoiceQuestion way
}
else if (q instanceof Question)
{
    // Do the task the Question way
}
```

In this case, you should implement a method *doTheTask* in the *Question* class, override it in *ChoiceQuestion*, and call

```
q.doTheTask();
```



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a program that demonstrates the *toString* method and the instanceof operator.



SELF CHECK

21. Why does the call

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

produce a result such as java.io.PrintStream@7a84e4?

22. Will the following code fragment compile? Will it run? If not, what error is reported?

```
Object obj = "Hello";
System.out.println(obj.length());
```

- 23.** Will the following code fragment compile? Will it run? If not, what error is reported?

```
Object obj = "Who was the inventor of Java?";
Question q = (Question) obj;
q.display();
```

- 24.** Why don't we simply store all objects in variables of type `Object`?

- 25.** Assuming that `x` is an object reference, what is the value of `x instanceof Object`?

Practice It Now you can try these exercises at the end of the chapter: E9.7, E9.11, E9.15.

Common Error 9.5



Don't Use Type Tests

Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if (q instanceof ChoiceQuestion) // Don't do this
{
    // Do the task the ChoiceQuestion way
}
else if (q instanceof Question)
{
    // Do the task the Question way
}
```

This is a poor strategy. If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
else if (q instanceof NumericQuestion)
{
    // Do the task the NumericQuestion way
}
```

In contrast, consider the addition of a class `NumericQuestion` to our quiz program. *Nothing* needs to change in that program because it uses polymorphism, not type tests.

Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead. Declare a method `doTheTask` in the superclass, override it in the subclasses, and call

```
q.doTheTask();
```

Special Topic 9.6



Inheritance and the `toString` Method

You just saw how to write a `toString` method: Form a string consisting of the class name and the names and values of the instance variables. However, if you want your `toString` method to be usable by subclasses of your class, you need to work a bit harder. Instead of hardcoding the class name, call the `getClass` method (which every class inherits from the `Object` class) to obtain an object that describes a class and its properties. Then invoke the `getName` method to get the name of the class:

```
public String toString()
{
    return getClass().getName() + "[balance=" + balance + "]";
}
```

Then the `toString` method prints the correct class name when you apply it to a subclass, say a `SavingsAccount`.

```
SavingsAccount mom'sSavings = . . . ;
System.out.println(mom'sSavings);
// Prints "SavingsAccount[balance=10000]"
```

Of course, in the subclass, you should override `toString` and add the values of the subclass instance variables. Note that you must call `super.toString` to get the instance variables of the superclass—the subclass can't access them directly.

```
public class SavingsAccount extends BankAccount
{
    . .
    public String toString()
    {
        return super.toString() + "[interestRate=" + interestRate + "]";
    }
}
```

Now a savings account is converted to a string such as `SavingsAccount[balance= 10000][interestRate=5]`. The brackets show which variables belong to the superclass.

Special Topic 9.7



Inheritance and the equals Method

You just saw how to write an `equals` method: Cast the `otherObject` parameter variable to the type of your class, and then compare the instance variables of the implicit parameter and the explicit parameter.

But what if someone called `stamp1.equals(x)` where `x` wasn't a `Stamp` object? Then the bad cast would generate an exception. It is a good idea to test whether `otherObject` really is an instance of the `Stamp` class. The easiest test would be with the `instanceof` operator. However, that test is not specific enough. It would be possible for `otherObject` to belong to some subclass of `Stamp`. To rule out that possibility, you should test whether the two objects belong to the same class. If not, return `false`.

```
if (getClass() != otherObject.getClass()) { return false; }
```

Moreover, the Java language specification demands that the `equals` method return `false` when `otherObject` is `null`.

Here is an improved version of the `equals` method that takes these two points into account:

```
public boolean equals(Object otherObject)
{
    if (otherObject == null) { return false; }
    if (getClass() != otherObject.getClass()) { return false; }
    Stamp other = (Stamp) otherObject;
    return color.equals(other.color) && value == other.value;
}
```

When you implement `equals` in a subclass, you should first call `equals` in the superclass to check whether the superclass instance variables match. Here is an example:

```
public CollectibleStamp extends Stamp
{
    private int year;
    .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) { return false; }
        CollectibleStamp other = (CollectibleStamp) otherObject;
        return year == other.year;
    }
}
```

9.6 Interface Types

It is often possible to design a general and reusable mechanism for processing objects by focusing on the essential operations that an algorithm needs. You use *interface types* to express these operations.

9.6.1 Defining an Interface

Consider the following method that computes the average balance in an array of `BankAccount` objects:

```
public static double average(BankAccount[] objects)
{
    if (objects.length == 0) { return 0; }
    double sum = 0;
    for (BankAccount obj : objects)
    {
        sum = sum + obj.getBalance();
    }
    return sum / objects.length;
}
```

Now suppose you have an array of `Country` objects and want to determine the average of the areas:

```
public static double average(Country[] objects)
{
    if (objects.length == 0) { return 0; }
    double sum = 0;
    for (Country obj : objects)
    {
        sum = sum + obj.getArea();
    }
    return sum / objects.length;
}
```

Clearly, the algorithm for computing the result is the same in both cases, but the details of measurement differ. How can we write a *single* method that computes the averages of both bank accounts and countries?

This standmixer provides the “rotation” service to any attachment that conforms to a common interface. Similarly, the average method at the end of this section works with any class that implements a common interface.



© gregory horler/Stockphoto.

Syntax 9.4 Interface Types

Syntax Declaring:

```
public interface InterfaceName
{
    method declarations
}
```

Implementing:

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
    instance variables
    methods
}
```

Interface methods are automatically public.

```
public interface Measurable
{
    double getMeasure();
}
```

Abstract methods have no implementation.

```
public class BankAccount implements Measurable
{
    . . .
    public double getMeasure()
    {
        return balance;
    }
}
```

A class can implement one or more interface types.

Implementation for the abstract method that was declared in the interface type.

Other BankAccount methods.

A Java interface type contains the return types, names, and parameter variables of a set of methods.

Unlike a class, an interface type provides no implementation.

Suppose that the classes agree on a single method `getMeasure` that obtains the measure to be used in the data analysis. For bank accounts, `getMeasure` returns the balance. For countries, `getMeasure` returns the area. Other classes can participate too, provided that their `getMeasure` method returns an appropriate value.

Then we can implement a single method that computes

```
sum = sum + obj.getMeasure();
```

What is the type of the variable `obj`? Any class that has a `getMeasure` method.

In Java, an **interface type** is used to specify required operations. We will declare an interface type that we call `Measurable`:

```
public interface Measurable
{
    double getMeasure();
}
```

The interface declaration lists all methods that the interface type requires. The `Measurable` interface type requires a single method, but in general, an interface type can require multiple methods. (Note that the `Measurable` type is not a type in the standard library—it is a type that was created specifically for this book.)

An interface type is similar to a class, but there are several important differences:

- Methods in an interface type must be *abstract* (that is, without an implementation) or, as of Java 8, static or default methods (see Java 8 Notes 9.1 and 9.2).
- All methods in an interface type are automatically public.
- An interface type cannot have instance variables.
- An interface type cannot have static methods.

By using an interface type for a parameter variable, a method can accept objects from many classes.

We can use the interface type `Measurable` to implement a “universal” method for computing averages:

```
public static double average(Measurable[] objects)
{
    if (objects.length == 0) { return 0; }
    double sum = 0;
    for (Measurable obj : objects)
    {
        sum = sum + obj.getMeasure();
    }
    return sum / objects.length;
}
```

9.6.2 Implementing an Interface

The `implements` reserved word indicates which interfaces a class implements.

The average method is usable for objects of any class that **implements** the `Measurable` interface. A class implements an interface type if it declares the interface in an `implements` clause. The class should implement the abstract method or methods that the interface requires. Let’s modify the `BankAccount` class to implement the `Measurable` interface.

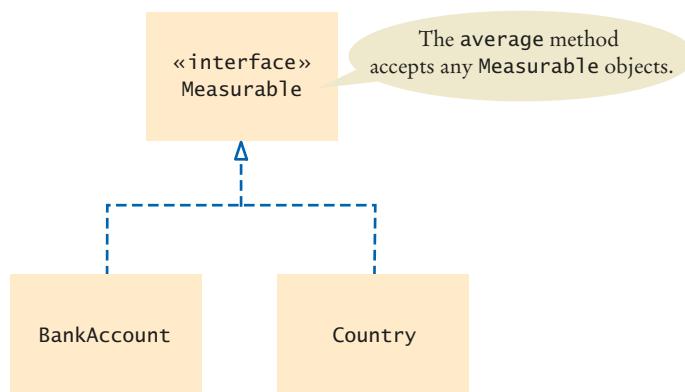
```
public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
    ...
}
```

Note that the class must declare the method as `public`, whereas the interface type need not—all methods in an interface type are `public`.

Similarly, it is an easy matter to implement a `Country` class that implements the `Measurable` interface.

```
public class Country implements Measurable
{
    public double getMeasure()
    {
        return area;
    }
    ...
}
```

Figure 11
Classes that Implement
the Measurable Interface



A reference to a `BankAccount` or `Country` can be converted to a `Measurable` reference. The sample program at the end of this section shows how the same average method can compute the average of a collection of bank accounts or countries.

In summary, the `Measurable` interface expresses what all measurable objects have in common. This commonality makes it possible to write methods such as `average` that are usable for many classes.

Figure 11 shows a diagram of the classes and interfaces in this program. A dotted arrow with a triangular tip denotes the “implements” relationship.

sec06/MeasurableDemo.java

```

1  /**
2   * This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasurableDemo
5  {
6      public static void main(String[] args)
7      {
8          Measurable[] accounts = new Measurable[3];
9          accounts[0] = new BankAccount(0);
10         accounts[1] = new BankAccount(10000);
11         accounts[2] = new BankAccount(2000);
12
13         System.out.println("Average balance: "
14             + average(accounts));
15
16         Measurable[] countries = new Measurable[3];
17         countries[0] = new Country("Uruguay", 176220);
18         countries[1] = new Country("Thailand", 514000);
19         countries[2] = new Country("Belgium", 30510);
20
21         System.out.println("Average area: "
22             + average(countries));
23     }
24
25
26     /**
27      * Computes the average of the measures of the given objects.
28      * @param objects an array of Measurable objects
29      * @return the average of the measures
30     */
31     public static double average(Measurable[] objects)
32     {
33         if (objects.length == 0) { return 0; }
34         double sum = 0;
35         for (Measurable obj : objects)
36         {
37             sum = sum + obj.getMeasure();
38         }
39         return sum / objects.length;
40     }
}

```

Program Run

```

Average balance: 4000.0
Average area: 240243.3333333334

```

9.6.3 The Comparable Interface

Implement the Comparable interface so that objects of your class can be compared, for example, in a sort method.

In the preceding sections, we defined the `Measurable` interface and provided an average method that works with any classes implementing that interface. In this section, you will learn about the `Comparable` interface of the standard Java library.

The `Measurable` interface is used for measuring a single object. The `Comparable` interface is more complex because comparisons involve two objects. The interface declares a `compareTo` method. The call

`a.compareTo(b)`

must return a negative number if `a` should come before `b`, zero if `a` and `b` are the same, and a positive number otherwise.

The `Comparable` interface has a single method:

```
public interface Comparable
{
    int compareTo(Object otherObject);
}
```

For example, the `BankAccount` class can implement `Comparable` like this:

```
public class BankAccount implements Comparable
{
    . . .
    public int compareTo(Object otherObject)
    {
        BankAccount other = (BankAccount) otherObject;
        if (balance < other.balance) { return -1; }
        if (balance > other.balance) { return 1; }
        return 0;
    }
    . . .
}
```

This `compareTo` method compares bank accounts by their balance. Note that the `compareTo` method has a parameter variable of type `Object`. To turn it into a `BankAccount` reference, we use a cast:

```
BankAccount other = (BankAccount) otherObject;
```

Once the `BankAccount` class implements the `Comparable` interface, you can sort an array of bank accounts with the `Arrays.sort` method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

The `accounts` array is now sorted by increasing balance.



The `compareTo` method checks whether another object is larger or smaller.

SELF CHECK

- 26.** Suppose you want to use the average method to find the average salary of Employee objects. What condition must the Employee class fulfill?
- 27.** Why can't the average method have a parameter variable of type Object[]?
- 28.** Why can't you use the average method to find the average length of String objects?
- 29.** What is wrong with this code?
- ```
Measurable meas = new Measurable();
System.out.println(meas.getMeasure());
```
- 30.** How can you sort an array of Country objects by increasing area?
- 31.** Can you use the Arrays.sort method to sort an array of String objects? Check the API documentation for the String class.

**Practice It**

Now you can try these exercises at the end of the chapter: R9.14, E9.18, E9.19.

**Common Error 9.6****Forgetting to Declare Implementing Methods as Public**

The methods in an interface are not declared as public, because they are public by default. However, the methods in a class are *not* public by default. It is a common error to forget the public reserved word when declaring a method from an interface:

```
public class BankAccount implements Measurable
{
 double getMeasure() // Oops—should be public
 {
 return balance;
 }
 ...
}
```

Then the compiler complains that the method has a weaker access level, namely package access instead of public access (see Special Topic 8.4). The remedy is to declare the method as public.

**Programming Tip 9.2****Comparing Integers and Floating-Point Numbers**

When you implement a comparison method, you need to return a negative integer to indicate that the first object should come before the other, zero if they are equal, or a positive integer otherwise. You have seen how to implement this decision with three branches. When you compare *nonnegative* integers, there is a simpler way: subtract the integers:

```
public class Person implements Comparable
{
 private int id; // Must be ≥ 0
 ...
 public int compareTo(Object otherObject)
 {
 Person other = (Person) otherObject;
 return id - other.id;
 }
}
```

The difference is negative if id < other.id, zero if the values are the same, and positive otherwise.

This trick doesn't work if the integers can be negative because the difference can overflow (see Exercise R9.15). However, the `Integer.compare` method always works:

```
return Integer.compare(id, other.id); // Safe for negative integers
```

You cannot compare floating-point values by subtraction (see Exercise R9.16). Instead, use the `Double.compare` method:

```
public class BankAccount implements Comparable
{
 .
 .
 public int compareTo(Object otherObject)
 {
 BankAccount other = (BankAccount) otherObject;
 return Double.compare(balance, other.balance);
 }
}
```

### Special Topic 9.8



### Constants in Interfaces

Interfaces cannot have instance variables, but it is legal to specify **constants**.

When declaring a constant in an interface, you can (and should) omit the reserved words `public static final`, because all variables in an interface are automatically `public static final`. For example,

```
public interface Measurable
{
 double OUNCES_PER_LITER = 33.814;
 .
}
```

To use this constant in your programs, add the interface name:

```
Measurable.OUNCES_PER_LITER
```

### Special Topic 9.9



### Generic Interface Types

In Section 9.6.3, you saw how to use the “raw” version of the `Comparable` interface type. In fact, the `Comparable` interface is a parameterized type, similar to the `ArrayList` type:

```
public interface Comparable<T>
{
 int compareTo(T other)
}
```

The type parameter specifies the type of the objects that this class is willing to accept for comparison. Usually, this type is the same as the class type itself. For example, the `BankAccount` class would implement `Comparable<BankAccount>`, like this:

```
public class BankAccount implements Comparable<BankAccount>
{
 .
 .
 public int compareTo(BankAccount other)
 {
 return Double.compare(balance, other.balance);
 }
}
```

The type parameter has a significant advantage: You need not use a cast to convert an `Object` parameter variable into the desired type.

Similarly, the `Measurer` interface can be improved by making it into a generic type:

```
public interface Measurer<T>
{
 double measure(T anObject);
}
```

The type parameter specifies the type of the parameter of the `measure` method. Again, you avoid the cast from `Object` when implementing the interface:

```
public class AreaMeasurer implements Measurer<Rectangle>
{
 public double measure(Rectangle anObject)
 {
 double area = anObject.getWidth() * anObject.getHeight();
 return area;
 }
}
```

(See Chapter 18 for an in-depth discussion of implementing and using generic classes.)

### Java 8 Note 9.1



**FULL CODE EXAMPLE**  
Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download an example of a static method in an interface.

### Static Methods in Interfaces

Before Java 8, all methods in an interface had to be abstract. Java 8 allows static methods in interfaces that work exactly like static methods in classes. A static method of an interface does not operate on objects, and its purpose should relate to the interface that contains it.

For example, it would be perfectly sensible to place the `average` method from Section 9.6.1 into the `Measurable` interface:

```
public interface Measurable
{
 double getMeasure(); // An abstract method
 static double average(Measurable[] objects) // A static method
 {
 . . . // Same implementation as in Section 9.6.1
 }
}
```

To call this method, provide the name of the interface and the method name:

```
double meanArea = Measurable.average(countries);
```

### Java 8 Note 9.2



### Default Methods

A **default method** is a non-static method in an interface that has an implementation. A class that implements the method either inherits the default behavior or overrides it. By providing default methods in an interface, it is less work to define a class that implements an interface.

For example, the `Measurable` interface can declare `getMeasure` as a default method:

```
public interface Measurable
{
 default double getMeasure() { return 0; }
}
```

If a class implements the interface and doesn't provide a `getMeasure` method, then it inherits this default method.

This particular example isn't all that useful. One doesn't normally want each object to have measure zero. Here is a more interesting example, in which a default method calls another interface method:

```
public interface Measurable
{
 double getMeasure(); // An abstract method
 default boolean smallerThan(Measurable other)
 {
 return getMeasure() < other.getMeasure();
 }
}
```

The `smallerThan` method tests whether an object has a smaller measure than another, which is useful for arranging objects by increasing measure.

A class that implements the `Measurable` interface only needs to implement `getMeasure`, and it automatically inherits the `smallerThan` method. This can be a very useful mechanism. For example, the `Comparator` interface that is described in Special Topic 14.5 has one abstract method but more than a dozen default methods.

## Special Topic 9.10



### Function Objects

In the preceding section, you saw how the `Measurable` interface type makes it possible to provide services that work for many classes—provided they are willing to implement the interface type. But what can you do if a class does not do so? For example, we might want to compute the average length of a collection of strings, but `String` does not implement `Measurable`.

Let's rethink our approach. The average method needs to measure each object. When the objects are required to be of type `Measurable`, the responsibility for measuring lies with the objects themselves, which is the cause of the limitation that we noted. It would be better if another object could carry out the measurement. Let's move the measurement method into a different interface:

```
public interface Measurer
{
 double measure(Object anObject);
}
```

The `measure` method measures an object and returns its measurement. We use a parameter variable of type `Object`, the “lowest common denominator” of all classes in Java, because we do not want to restrict which classes can be measured.

We add a parameter variable of type `Measurer` to the average method:

```
public static double average(Object[] objects, Measurer meas)
{
 if (objects.length == 0) { return 0; }
 double sum = 0;
 for (Object obj : objects)
 {
 sum = sum + meas.measure(obj);
 }
 return sum / objects.length;
}
```

When calling the method, you need to supply a `Measurer` object. That is, you need to implement a class with a `measure` method, and then create an object of that class. Let's do that for measuring strings:

```
public class StringMeasurer implements Measurer
{
 public double measure(Object obj)
 {
 String str = (String) obj; // Cast obj to String type
 return str.length();
 }
}
```

Note that the `measure` method must accept an argument of type `Object`, even though this particular measurer just wants to measure strings. The parameter variable must have the same type as in the `Measurer` interface. Therefore, the `Object` parameter variable is cast to the `String` type.

Finally, we are ready to compute the average length of an array of strings:

```
String[] words = { "Mary", "had", "a", "little", "lamb" };
Measurer lengthMeasurer = new StringMeasurer();
double result = average(words, lengthMeasurer); // result is set to 3.6
```

An object such as `lengthMeasurer` is called a *function object*. The sole purpose of the object is to execute a single method, in our case `measure`. (In mathematics, as well as many other programming languages, the term “function” is used where Java uses “method”.)

The `Comparator` interface, discussed in Special Topic 14.4, is another example of an interface for function objects.

## Java 8 Note 9.3

8

### Lambda Expressions

In Special Topic 9.10, you saw how to use function objects for specifying variations in behavior. The `average` method needs to measure each object, and it does so by calling the `measure` method of the supplied `Measurer` object.

Unfortunately, the caller of the `average` method has to do a fair amount of work; namely, to define a class that implements the `Measurer` interface and to construct an object of that class. Java 8 has a convenient shortcut for these steps, provided that the interface has a *single abstract method*. Such an interface is called a **functional interface** because its purpose is to define a single function. The `Measurer` interface is an example of a functional interface.

To specify that single function, you can use a **lambda expression**, an expression that defines the parameters and return value of a method in a compact notation. Here is an example:

```
(Object obj) -> ((BankAccount) obj).getBalance()
```

This expression defines a function that, given an object, casts it to a `BankAccount` and returns the balance.

(The term “lambda expression” comes from a mathematical notation that uses the Greek letter lambda ( $\lambda$ ) instead of the `->` symbol. In other programming languages, such an expression is called a *function expression*.)

A lambda expression cannot stand alone. It needs to be assigned to a variable whose type is a functional interface:

```
Measurer accountMeas = (Object obj) -> ((BankAccount) obj).getBalance();
```

Now the following actions occur:

1. A class is defined that implements the functional interface. The single abstract method is defined by the lambda expression.
2. An object of that class is constructed.
3. The variable is assigned a reference to that object.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download a complete program that demonstrates the string measurer.

You can also pass a lambda expression to a method. Then the parameter variable of the method is initialized with the constructed object. For example, consider the call

```
double averageBalance = average(accounts,
 (Object obj) -> ((BankAccount) obj).getBalance());
```

In the same way as before, an object is constructed that belongs to a class implementing `Measurer`. The object is used to initialize the parameter variable `meas` of the `average` method. Recall that the parameter variable has type `Measurer`:

```
public static double average(Object[] objects, Measurer meas)
{
 . . .
 sum = sum + meas.measure(obj);
 . . .
}
```

The `average` method calls the `measure` method on `meas`, which in turn executes the body of the lambda expression.

In its simplest form, a lambda expression contains a list of parameters and the expression that is being computed from the parameters. If more work needs to be done, you can write a method body in the usual way, enclosed in braces and with a `return` statement:

```
Measurer areaMeas = (Object obj) ->
{
 Rectangle r = (Rectangle) obj;
 return r.getWidth() * r.getHeight();
};
```

Lambda expressions enable the caller of a method to provide code that is called inside the method, and they enable the implementor of the method to invoke that code as needed. This can be achieved as follows:

1. The implementor of the method defines an interface that describes the purpose of the code to be executed. That interface has a single method.
2. The method receives a parameter of that interface, and calls the single method of the interface whenever the code that can vary needs to be called.
3. The caller of the method provides a lambda expression whose body is the code that should be called in this invocation.

You will see additional examples of using lambda expressions for event handlers (Java 8 Note 10.1) and comparators (Section 14.8). Lambda expressions are extensively used in the “streams” API for processing large data sets.



### WORKED EXAMPLE 9.2



### Investigating Number Sequences

Learn how to use a `Sequence` interface to investigate properties of arbitrary number sequences. Go to [wiley.com/go/bj102examples](http://wiley.com/go/bj102examples) and download Worked Example 9.2.



© Norebbo/  
iStockphoto.



### VIDEO EXAMPLE 9.2

### Drawing Geometric Shapes

In this Video Example, you will see how to use inheritance to describe and draw different geometric shapes. Go to [wiley.com/go/bj102videos](http://wiley.com/go/bj102videos) to view Video Example 9.2.





## Computing & Society 9.1 Who Controls the Internet?

In 1962, J.C.R. Licklider was head of the first computer research program at DARPA, the Defense Advanced Research Projects Agency. He wrote a series of papers describing a “galactic network” through which computer users could access data and programs from other sites. This was well before computer networks were invented. By 1969, four computers—three in California and one in Utah—were connected to the ARPANET, the precursor of the Internet. The network grew quickly, linking computers at many universities and research organizations. It was originally thought that most network users wanted to run programs on remote computers. Using remote execution, a researcher at one institution would be able to access an underutilized computer at a different site. It quickly became apparent that remote execution was not what the network was actually used for. Instead, the “killer application” was electronic mail: the transfer of messages between computer users at different locations.

In 1972, Bob Kahn proposed to extend ARPANET into the *Internet*: a collection of interoperable networks. All networks on the Internet share common *protocols* for data transmission. Kahn and Vinton Cerf developed a protocol, now called TCP/IP (Transmission Control Protocol/Internet Protocol). On January 1, 1983, all hosts on the Internet simultaneously switched to the TCP/IP protocol (which is used to this day).

Over time, researchers, computer scientists, and hobbyists published increasing amounts of information on the Internet. For example, Project Gutenberg makes available the text of important classical books, whose copyright has expired, in computer-readable form ([www.gutenberg.org](http://www.gutenberg.org)). In 1989, Tim Berners-Lee, a computer scientist at CERN (the European organization for nuclear research) started work on hyperlinked documents, allowing users to browse by following links to related documents. This infrastructure is now known as the World Wide Web.

The first interfaces to retrieve this information were, by today’s standards, unbelievably clumsy and hard to use. In March 1993, WWW traffic was 0.1 percent of all Internet traffic. All that changed when Marc Andreessen, then a graduate student working for the National Center for Supercomputing Applications (NCSA), released Mosaic. Mosaic displayed web pages in graphical form, using images, fonts, and colors (see the figure). Andreessen went on to fame and fortune at Netscape, and Microsoft licensed the Mosaic code to create Internet Explorer. By 1996, WWW traffic accounted for more than half of the data transported on the Internet.

The Internet has a very democratic structure. Anyone can publish anything, and anyone can read whatever has been published. This does not always sit well with governments and corporations.

Many governments control the Internet infrastructure in their country. For example, an Internet user in

China, searching for the Tiananmen Square massacre or air pollution in their hometown, may find nothing. Vietnam blocks access to Facebook, perhaps fearing that anti-government protesters might use it to organize themselves. The U.S. government has required publicly funded libraries and schools to install filters that block sexually-explicit and hate speech, and its security organizations have spied on the Internet usage of citizens.

When the Internet is delivered by phone or TV cable companies, those companies sometimes interfere with competing Internet offerings. Cell phone companies refused to carry Voice-over-IP services, and cable companies slowed down movie streaming.

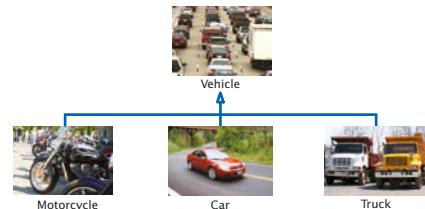
The Internet has become a powerful force for delivering information—both good and bad. It is our responsibility as citizens to demand of our government that we can control which information to access.

The NCSA Mosaic Browser

## CHAPTER SUMMARY

### Explain the notions of inheritance, superclass, and subclass.

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object.



### Implement subclasses in Java.

- A subclass inherits all methods that it does not override.
- A subclass can override a superclass method by providing a new implementation.
- The `extends` reserved word indicates that a class inherits from a superclass.



### Implement methods that override methods from a superclass.

- An overriding method can extend or replace the functionality of the superclass method.
- Use the reserved word `super` to call a superclass method.
- Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.
- To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.
- The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

### Use polymorphism for processing objects of related types.

- A subclass reference can be used when a superclass reference is expected.
- Polymorphism (“having multiple shapes”) allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
- An abstract method is a method whose implementation is not specified.
- An abstract class is a class that cannot be instantiated.



### Use the `toString` method and `instanceof` operator with objects.

- Override the `toString` method to yield a string that describes the object’s state.
- The `equals` method checks whether two objects have the same contents.
- If you know that an object belongs to a given class, use a cast to convert the type.
- The `instanceof` operator tests whether an object belongs to a particular type.

**Use interface types for algorithms that process objects of different classes.**

- A Java interface type contains the return types, names, and parameter variables of a set of methods.
- Unlike a class, an interface type provides no implementation.
- By using an interface type for a parameter variable, a method can accept objects from many classes.
- The `implements` reserved word indicates which interfaces a class implements.
- Implement the `Comparable` interface so that objects of your class can be compared, for example, in a `sort` method.

**REVIEW EXERCISES**

- **R9.1** Identify the superclass and subclass in each of the following pairs of classes.
- a.** Employee, Manager
  - b.** GraduateStudent, Student
  - c.** Person, Student
  - d.** Employee, Professor
  - e.** BankAccount, CheckingAccount
  - f.** Vehicle, Car
  - g.** Vehicle, Minivan
  - h.** Car, Minivan
  - i.** Truck, Vehicle
- **R9.2** Consider a program for managing inventory in a small appliance store. Why isn't it useful to have a superclass `SmallAppliance` and subclasses `Toaster`, `CarVacuum`, `TravelIron`, and so on?
- **R9.3** Which methods does the `ChoiceQuestion` class inherit from its superclass? Which methods does it override? Which methods does it add?
- **R9.4** Which methods does the `SavingsAccount` class in How To 9.1 inherit from its superclass? Which methods does it override? Which methods does it add?
- **R9.5** List the instance variables of a `CheckingAccount` object from How To 9.1.
- ■ **R9.6** Suppose the class `Sub` extends the class `Sandwich`. Which of the following assignments are legal?
- ```
Sandwich x = new Sandwich();
Sub y = new Sub();
```
- a.** `x = y;`
 - b.** `y = x;`
 - c.** `y = new Sandwich();`
 - d.** `x = new Sub();`
- **R9.7** Draw an inheritance diagram that shows the inheritance relationships between these classes.
- | | |
|---|---|
| <ul style="list-style-type: none"> • Person • Employee • Student | <ul style="list-style-type: none"> • Instructor • Classroom • Object |
|---|---|

- **R9.8** In an object-oriented traffic simulation system, we have the classes listed below. Draw an inheritance diagram that shows the relationships between these classes.

- Vehicle
- Car
- Truck
- Sedan
- Coupe
- PickupTruck
- SportUtilityVehicle
- Minivan
- Bicycle
- Motorcycle

- **R9.9** What inheritance relationships would you establish among the following classes?

- Student
- Professor
- TeachingAssistant
- Employee
- Secretary
- DepartmentChair
- Janitor
- SeminarSpeaker
- Person
- Course
- Seminar
- Lecture
- ComputerLab

- ■ **R9.10** How does a cast such as (BankAccount) x differ from a cast of number values such as (int) x?

- ■ ■ **R9.11** Which of these conditions returns true? Check the Java documentation for the inheritance patterns. Recall that `System.out` is an object of the `PrintStream` class.

- `System.out instanceof PrintStream`
- `System.out instanceof OutputStream`
- `System.out instanceof LogStream`
- `System.out instanceof Object`
- `System.out instanceof Closeable`
- `System.out instanceof Writer`

- ■ **R9.12** Suppose C is a class that implements the interfaces I and J. Which of the following assignments require a cast?

```
C c = . . .;
I i = . . .;
J j = . . .;
```

- `c = i;`
- `j = c;`
- `i = j;`

- ■ **R9.13** Suppose C is a class that implements the interfaces I and J, and i is declared as

```
I i = new C();
```

Which of the following statements will throw an exception?

- `C c = (C) i;`
- `J j = (J) i;`
- `i = (I) null;`

- ■ **R9.14** Suppose the class Sandwich implements the Edible interface, and you are given the variable declarations

```
Sandwich sub = new Sandwich();
Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
Edible e = null;
```

Which of the following assignment statements are legal?

- a. e = sub;
- b. sub = e;
- c. sub = (Sandwich) e;
- d. sub = (Sandwich) cerealBox;
- e. e = cerealBox;
- f. e = (Edible) cerealBox;
- g. e = (Rectangle) cerealBox;
- h. e = (Rectangle) null;

- R9.15 Suppose an int value a is two billion and b is -a. What is the result of a - b? Of b - a? What is the result of Integer.compare(a, b)? Of Integer.compare(b - a)?
- R9.16 Suppose a double value a is 0.6 and b is 0.3. What is the result of (int)(a - b)? Of (int)(b - a)? What is the result of Double.compare(a, b)? Of Double.compare(b - a)?

PRACTICE EXERCISES

- E9.1 Add a class NumericQuestion to the question hierarchy of Section 9.1. If the response and the expected answer differ by no more than 0.01, then accept the response as correct.
- E9.2 Add a class FillInQuestion to the question hierarchy of Section 9.1. Such a question is constructed with a string that contains the answer, surrounded by ___, for example, "The inventor of Java was ___James Gosling___. The question should be displayed as
The inventor of Java was _____
- E9.3 Modify the checkAnswer method of the Question class so that it does not take into account different spaces or upper/lowercase characters. For example, the response "JAMES gosling" should match an answer of "James Gosling".
- E9.4 Add a class AnyCorrectChoiceQuestion to the question hierarchy of Section 9.1 that allows multiple correct choices. The respondent should provide any one of the correct choices. The answer string should contain all of the correct choices, separated by spaces. Provide instructions in the question text.
- E9.5 Add a class MultiChoiceQuestion to the question hierarchy of Section 9.1 that allows multiple correct choices. The respondent should provide all correct choices, separated by spaces. Provide instructions in the question text.
- E9.6 Add a method addText to the Question superclass and provide a different implementation of ChoiceQuestion that calls addText rather than storing an array list of choices.
- E9.7 Provide `toString` methods for the Question and ChoiceQuestion classes.
- E9.8 Implement a subclass of BankAccount from How To 9.1 called BasicAccount whose withdraw method will not withdraw more money than is currently in the account.
- E9.9 Implement a subclass of BankAccount from How To 9.1 called BasicAccount whose withdraw method charges a penalty of \$30 for each withdrawal that results in an overdraft.

- E9.10** Reimplement the `CheckingAccount` class from How To 9.1 so that the first overdraft in any given month incurs a \$20 penalty, and any further overdrafts in the same month result in a \$30 penalty.
- E9.11** Implement a superclass `Person`. Make two classes, `Student` and `Instructor`, that inherit from `Person`. A person has a name and a year of birth. A student has a major, and an instructor has a salary. Write the class declarations, the constructors, and the methods `toString` for all classes. Supply a test program that tests these classes and methods.
- E9.12** Make a class `Employee` with a name and salary. Make a class `Manager` inherit from `Employee`. Add an instance variable, named `department`, of type `String`. Supply a method `toString` that prints the manager's name, department, and salary. Make a class `Executive` inherit from `Manager`. Supply appropriate `toString` methods for all classes. Supply a test program that tests these classes and methods.
- E9.13** The `Rectangle` class of the standard Java library does not supply a method to compute the area or the perimeter of a rectangle. Provide a subclass `BetterRectangle` of the `Rectangle` class that has `getPerimeter` and `getArea` methods. *Do not add any instance variables*. In the constructor, call the `setLocation` and `setSize` methods of the `Rectangle` class. Provide a program that tests the methods that you supplied.
- E9.14** Repeat Exercise E9.13, but in the `BetterRectangle` constructor, invoke the superclass constructor.
- E9.15** A labeled point has *x*- and *y*-coordinates and a string label. Provide a class `LabeledPoint` with a constructor `LabeledPoint(int x, int y, String label)` and a `toString` method that displays *x*, *y*, and the label.
- E9.16** Reimplement the `LabeledPoint` class of Exercise E9.15 by storing the location in a `java.awt.Point` object. Your `toString` method should invoke the `toString` method of the `Point` class.
- E9.17** Modify the `SodaCan` class of Exercise P8.5 to implement the `Measurable` interface. The measure of a soda can should be its surface area. Write a program that computes the average surface area of an array of soda cans.
- E9.18** A person has a name and a height in centimeters. Use the `average` method in Section 9.6 to process a collection of `Person` objects.
- E9.19** Write a method

```
public static Measurable maximum(Measurable[] objects)
```

 that returns the object with the largest measure. Use that method to determine the country with the largest area from an array of countries.
- E9.20** Add static methods `largest` and `smallest` to the `Measurable` interface. The methods should return the object with the largest or smallest measure from an array of `Measurable` objects.
- E9.21** In the `Sequence` interface of Worked Example 9.2, add static methods that yield `Sequence` instances:

```
static Sequence multiplesOf(int n)
static Sequence powersOf(int n)
```

 For example, `Sequence.powersOf(2)` should return the same sequence as the `SquareSequence` class in the worked example.



- E9.22** In Worked Example 9.2, add a default method

```
default int[] values(int n)
```

that yields an array of the first n values of the sequence.



- E9.23** In Worked Example 9.2, make the process method a default method of the Sequence interface.

PROGRAMMING PROJECTS

- P9.1** Implement a class Clock whose getHours and getMinutes methods return the current time at your location. (Call `java.time.LocalTime.now().toString()` or, if you are not using Java 8, `new java.util.Date().toString()` and extract the time from that string.) Also provide a getTime method that returns a string with the hours and minutes by calling the getHours and getMinutes methods. Provide a subclass WorldClock whose constructor accepts a time offset. For example, if you live in California, a new `WorldClock(3)` should show the time in New York, three time zones ahead. Which methods did you override? (You should not override `getTime`.)

- P9.2** Add an alarm feature to the Clock class of Exercise P9.1. When `setAlarm(hours, minutes)` is called, the clock stores the alarm. When you call `getTime`, and the alarm time has been reached or exceeded, return the time followed by the string "Alarm" (or, if you prefer, the string "\u23F0") and clear the alarm. What do you need to do to make the `setAlarm` method work for WorldClock objects?

- P9.3** The `System.out.printf` method has predefined formats for printing integers, floating-point numbers, and other data types. But it is also extensible. If you use the `s` format, you can print any class that implements the `Formattable` interface. That interface has a single method:

```
void formatTo(Formatter formatter, int flags, int width, int precision)
```

In this exercise, you should make the `BankAccount` class implement the `Formattable` interface. Ignore the flags and precision and simply format the bank balance, using the given width. In order to achieve this task, you need to get an `Appendable` reference like this:

```
Appendable a = formatter.out();
```

`Appendable` is another interface with a method

```
void append(CharSequence sequence)
```

`CharSequence` is yet another interface that is implemented by (among others) the `String` class. Construct a string by first converting the bank balance into a string and then padding it with spaces so that it has the desired width. Pass that string to the `append` method.

- P9.4** Enhance the `formatTo` method of Exercise P9.3 by taking into account the precision.

- Business P9.5** Change the `CheckingAccount` class in How To 9.1 so that a \$1 fee is levied for deposits or withdrawals in excess of three free monthly transactions. Place the code for computing the fee into a separate method that you call from the `deposit` and `withdraw` methods.

- Business P9.6** Implement a superclass `Appointment` and subclasses `Onetime`, `Daily`, and `Monthly`. An appointment has a description (for example, “see the dentist”) and a date. Write a method `occursOn(int year, int month, int day)` that checks whether the appointment occurs on that date. For example, for a monthly appointment, you must check whether the day of the month matches. Then fill an array of `Appointment` objects with a mixture of appointments. Have the user enter a date and print out all appointments that occur on that date.

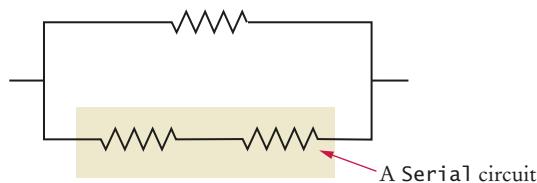


© Pali Rao/iStockphoto.

- Business P9.7** Improve the appointment book program of Exercise P9.6. Give the user the option to add new appointments. The user must specify the type of the appointment, the description, and the date.

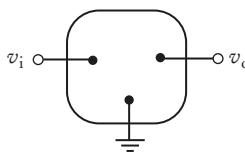
- Business P9.8** Improve the appointment book program of Exercises P9.6 and P9.7 by letting the user save the appointment data to a file and reload the data from a file. The saving part is straightforward: Make a method `save`. Save the type, description, and date to a file. The loading part is not so easy. First determine the type of the appointment to be loaded, create an object of that type, and then call a `load` method to load the data.

- Science P9.9** In this problem, you will model a circuit consisting of an arbitrary configuration of resistors. Provide a superclass `Circuit` with a instance method `getResistance`. Provide a subclass `Resistor` representing a single resistor. Provide subclasses `Serial` and `Parallel`, each of which contains an `ArrayList<Circuit>`. A `Serial` circuit models a series of circuits, each of which can be a single resistor or another circuit. Similarly, a `Parallel` circuit models a set of circuits in parallel. For example, the following circuit is a `Parallel` circuit containing a single resistor and one `Serial` circuit:

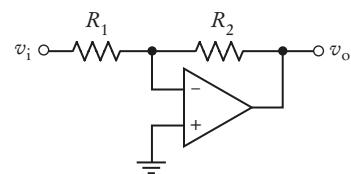


Use Ohm's law to compute the combined resistance.

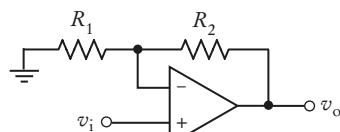
- Science P9.10** Part (a) of the figure below shows a symbolic representation of an electric circuit called an *amplifier*. The input to the amplifier is the voltage v_i and the output is the voltage v_o . The output of an amplifier is proportional to the input. The constant of proportionality is called the “gain” of the amplifier. Parts (b), (c), and (d) show schematics of three specific types of amplifier: the *inverting amplifier*, *noninverting amplifier*, and *voltage divider amplifier*. Each of these three amplifiers consists of two resistors and an op amp. The value of the gain of each amplifier depends on the values of its resistances. In particular, the gain, g , of the inverting amplifier is given by $g = -\frac{R_2}{R_1}$. Similarly the gains of the noninverting amplifier and voltage divider amplifier are given by $g = 1 + \frac{R_2}{R_1}$ and $g = \frac{R_2}{R_1 + R_2}$, respectively.



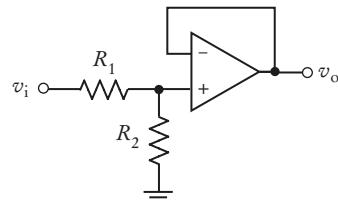
(a) Amplifier



(b) Inverting amplifier



(c) Noninverting amplifier



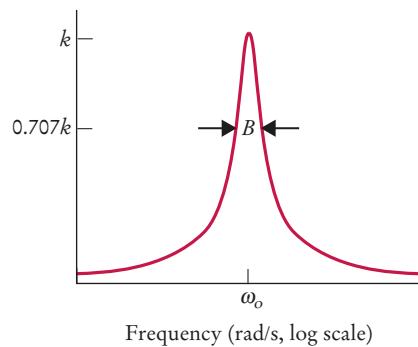
(d) Voltage divider amplifier

Write a Java program that represents the amplifier as a superclass and represents the inverting, noninverting, and voltage divider amplifiers as subclasses. Give the subclass two methods, `getGain` and a `getDescription` method that returns a string identifying the amplifier. Each subclass should have a constructor with two arguments, the resistances of the amplifier.

The subclasses need to override the `getGain` and `getDescription` methods of the superclass.

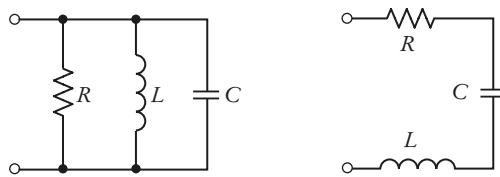
Supply a class that demonstrates that the subclasses all work properly for sample values of the resistances.

•• Science P9.11 Resonant circuits are used to select a signal (e.g., a radio station or TV channel) from among other competing signals. Resonant circuits are characterized by the frequency response shown in the figure below. The resonant frequency response is completely described by three parameters: the resonant frequency, ω_0 , the bandwidth, B , and the gain at the resonant frequency, k .



Frequency (rad/s, log scale)

Two simple resonant circuits are shown in the figure on the next page. The circuit in (a) is called a *parallel resonant circuit*. The circuit in (b) is called a *series resonant circuit*. Both resonant circuits consist of a resistor having resistance R , a capacitor having capacitance C , and an inductor having inductance L .



(a) Parallel resonant circuit

(b) Series resonant circuit

These circuits are designed by determining values of R , C , and L that cause the resonant frequency response to be described by specified values of ω_0 , B , and k . The design equations for the parallel resonant circuit are:

$$R = k, \quad C = \frac{1}{BR}, \text{ and} \quad L = \frac{1}{\omega_0^2 C}$$

Similarly, the design equations for the series resonant circuit are:

$$R = \frac{1}{k}, \quad L = \frac{R}{B}, \text{ and} \quad C = \frac{1}{\omega_0^2 L}$$

Write a Java program that represents `ResonantCircuit` as a superclass and represents the `SeriesResonantCircuit` and `ParallelResonantCircuit` as subclasses. Give the superclass three private instance variables representing the parameters ω_0 , B , and k of the resonant frequency response. The superclass should provide public instance methods to get and set each of these variables. The superclass should also provide a `display` method that prints a description of the resonant frequency response.

Each subclass should provide a method that designs the corresponding resonant circuit. The subclasses should also override the `display` method of the superclass to print descriptions of both the frequency response (the values of ω_0 , B , and k) and the circuit (the values of R , C , and L).

All classes should provide appropriate constructors.

Supply a class that demonstrates that the subclasses all work properly.

- ■ **P9.12** Modify the `display` method of the `LastDigitDistribution` class of Worked Example 9.2 so that it produces a histogram, like this:

```
0: ****
1: *****
2: *****
```

Scale the bars so that widest one has length 40.

- ■ **P9.13** Write a class `PrimeSequence` that implements the `Sequence` interface of Worked Example 9.2, and produces the sequence of prime numbers.

- **P9.14** Add a method `hasNext` to the `Sequence` interface of Worked Example 9.2 that returns `false` if the sequence has no more values. Implement a class `MySequence` producing a sequence of real data of your choice, such as populations of cities or countries, temperatures, or stock prices. Obtain the data from the Internet and reformat the values so that they are placed into an array. Return one value at a time in the `next` method, until you reach the end of the data. Your `SequenceDemo` class should display the distribution of the last digits of all sequence values.

- **P9.15** Provide a class `FirstDigitDistribution` that works just like the `LastDigitDistribution` class of Worked Example 9.2, except that it counts the distribution of the first digit of each value. (It is a well-known fact that the first digits of random values are *not* uniformly distributed. This fact has been used to detect accounting fraud, when sequences of transaction amounts had an unnatural distribution of their first digits.)

- **P9.16** Declare an interface `Filter` as follows:

```
public interface Filter { boolean accept(Object x); }
```

Add a `Filter` parameter to the `average` method of Section 9.6. Only objects that the filter accepts should be processed. Demonstrate your modification by processing a collection of bank accounts, filtering out all accounts with balances less than \$1,000.

- **P9.17** Solve Exercise P9.16, using a lambda expression for the filter.



- **P9.18** In Exercise P9.16, add a method to the `Filter` interface that counts how many objects are accepted by the filter:

```
static int count(Object[] values, Filter condition)
```



- **P9.19** In Exercise P9.16, add a method to the `Filter` interface that retains all objects accepted by the filter and removes the others:

```
static void retainAll(Object[] values, Filter condition)
```



- **P9.20** In Exercise P9.16, add a method `default boolean reject(Object x)` to the `Filter` interface that returns true for all objects that this filter doesn't accept.



- **P9.21** In Exercise P9.16, add a method `default Filter invert()` to the `Filter` interface that yields a filter accepting exactly the objects that this filter rejects.

- **P9.22** Consider an interface

```
public interface NumberFormatter
{
    String format(int n);
}
```

Provide four classes that implement this interface. A `DefaultFormatter` formats an integer in the usual way. A `DecimalSeparatorFormatter` formats an integer with decimal separators; for example, one million as 1,000,000. An `AccountingFormatter` formats negative numbers with parentheses; for example, -1 as (1). A `BaseFormatter` formats the number in base n , where n is any number between 2 and 36 that is provided in the constructor.

- **P9.23** Write a method that takes an array of integers and a `NumberFormatter` object (from Exercise P9.22) and prints each number on a separate line, formatted with the given formatter. The numbers should be right aligned.

ANSWERS TO SELF-CHECK QUESTIONS

1. Because every manager is an employee but not the other way around, the Manager class is more specialized. It is the subclass, and Employee is the superclass.
2. CheckingAccount and SavingsAccount both inherit from the more general class BankAccount.
3. Exception, Throwable
4. Vehicle, truck, motorcycle
5. It shouldn't. A quiz isn't a question; it *has* questions.
6. a, b, d
7.

```
public class Manager extends Employee
{
    private double bonus;
    // Constructors and methods omitted
}
```
8. name, baseSalary, and bonus
9.

```
public class Manager extends Employee
{
    ...
    public double getSalary() { . . . }
}
```
10. getName, setName, setBaseSalary
11. The method is not allowed to access the instance variable text from the superclass.
12. The type of the this reference is ChoiceQuestion. Therefore, the display method of ChoiceQuestion is selected, and the method calls itself.
13. Because there is no ambiguity. The subclass doesn't have a setAnswer method.
14.

```
public String getName()
{
    return "*" + super.getName();
}
```
15.

```
public double getSalary()
{
    return super.getSalary() + bonus;
}
```
16. a and c.
17. It belongs to the class BankAccount or one of its subclasses.
18. `Question[] quiz = new Question[SIZE];`
19. You cannot tell from the fragment—cq may be initialized with an object of a subclass of ChoiceQuestion. The display method of whatever object cq references is invoked.
20. No. This is a static method of the Math class. There is no implicit parameter object that could be used to dynamically look up a method.
21. Because the implementor of the PrintStream class did not supply a `toString` method.
22. The second line will not compile. The class Object does not have a method `length`.
23. The code will compile, but the second line will throw a class cast exception because Question is not a subclass of String.
24. There are only a few methods that can be invoked on variables of type Object.
25. The value is false if x is null and true otherwise.
26. It must implement the Measurable interface and provide a `getMeasure` method returning the salary.
27. The Object class doesn't have a `getMeasure` method.
28. You cannot modify the String class to implement Measurable—it is a library class. See Special Topic 9.10 for a solution.
29. Measurable is not a class. You cannot construct objects of type Measurable.
30. Have the Country class implement the Comparable interface, as shown below, and call Arrays.sort.


```
public class Country implements Comparable
{
    ...
    public int compareTo(Object otherObject)
    {
        Country other = (Country) otherObject;
        if (area < other.area) return -1;
        if (area > other.area) return 1;
        return 0;
    }
}
```
31. Yes, you can, because String implements the Comparable interface type.

WORKED EXAMPLE 9.1

Implementing an Employee Hierarchy for Payroll Processing

Problem Statement Your task is to implement payroll processing for different kinds of employees.

- Hourly employees get paid an hourly rate, but if they work more than 40 hours per week, the excess is paid at “time and a half”.
- Salaried employees get paid their salary, no matter how many hours they work.
- Managers are salaried employees who get paid a salary and a bonus.

Your program should compute the pay for a collection of employees. For each employee, ask for the number of hours worked in a given week, then display the wages earned.



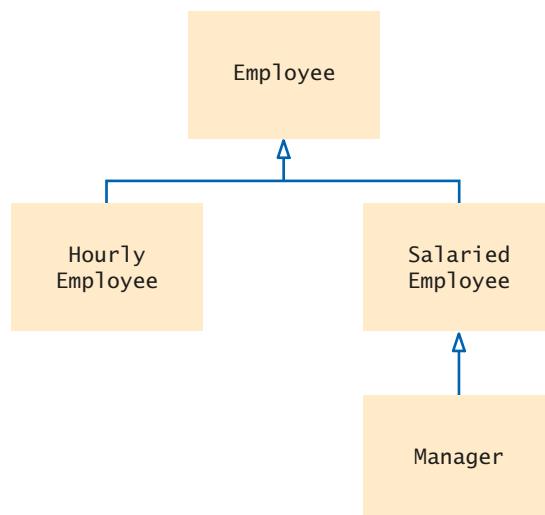
Jose Luis Pelaez Inc./
Getty Images Inc.

Step 1 List the classes that are part of the hierarchy.

In our case, the problem description lists three classes: `HourlyEmployee`, `SalariedEmployee`, and `Manager`. We need a class that expresses the commonality among them: `Employee`.

Step 2 Organize the classes into an inheritance hierarchy.

Here is the inheritance diagram for our classes.



Step 3 Determine the common responsibilities of the classes.

In order to discover the common responsibilities, write pseudocode for processing the objects.

For each employee
Print the name of the employee.
Read the number of hours worked.
Compute the wages due for those hours.

We conclude that the `Employee` superclass has these responsibilities:

Get the name.
Compute the wages due for a given number of hours.

WE2 Chapter 9 Inheritance and Interfaces

Step 4 Decide which methods are overridden in subclasses.

In our example, there is no variation in getting the employee's name, but the salary is computed differently in each subclass, so `weeklyPay` will be overridden in each subclass.

```
/***
 * An employee with a name and a mechanism for computing weekly pay.
 */
public class Employee
{
    . . .

    /**
     * Gets the name of this employee.
     * @return the name
     */
    public String getName() { . . . }

    /**
     * Computes the pay for one week of work.
     * @param hoursWorked the number of hours worked in the week
     * @return the pay for the given number of hours
     */
    public double weeklyPay(int hoursWorked) { . . . }
}
```

Step 5 Declare the public interface of each class.

We will construct employees by supplying their name and salary information.

```
public class HourlyEmployee extends Employee
{
    . . .

    /**
     * Constructs an hourly employee with a given name and weekly wage.
     */
    public HourlyEmployee(String name, double wage) { . . . }

    . . .

    public class SalariedEmployee extends Employee
    {
        . . .

        /**
         * Constructs a salaried employee with a given name and annual salary.
         */
        public SalariedEmployee(String name, double salary) { . . . }

        . . .

        public class Manager extends SalariedEmployee
        {
            . . .

            /**
             * Constructs a manager with a given name, annual salary and weekly bonus.
             */
            public Manager(String name, double salary, double bonus) { . . . }

            . . .
        }
    }
}
```

These constructors need to set the name of the `Employee` object. We will add a method `setName` to the `Employee` class for this purpose:

```
public class Employee
{
    . . .
    public void setName(String employeeName) { . . . }
    . . .
}
```

Of course, each subclass needs a method for computing the weekly wages:

```
// This method overrides the superclass method
public double weeklyPay(int hoursWorked) { . . . }
```

In this simple example, no further methods are required.

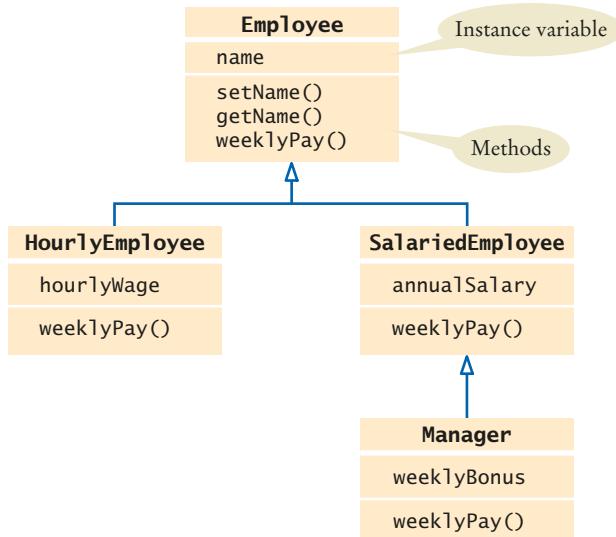
Step 6

Identify instance variables.

All employees have a name. Therefore, the `Employee` class should have an instance variable `name`. (See the revised hierarchy below.)

What about the salaries? Hourly employees have an hourly wage, whereas salaried employees have an annual salary. While it would be possible to store these values in an instance variable of the superclass, it would not be a good idea. The resulting code, which would need to make sense of what that number means, would be complex and error-prone.

Instead, `HourlyEmployee` objects will store the hourly wage and `SalariedEmployee` objects will store the annual salary. Manager objects need to store the weekly bonus.



Step 7

Implement constructors and methods.

In a subclass constructor, we need to remember to set the instance variables of the superclass:

```
public SalariedEmployee(String name, double salary)
{
    setName(name);
    annualSalary = salary;
}
```

Here we use a method. Special Topic 9.1 shows how to invoke a superclass constructor. We use that technique in the `Manager` constructor:

```
public Manager(String name, double salary, double bonus)
{
    super(name, salary)
    weeklyBonus = bonus;
```

WE4 Chapter 9 Inheritance and Interfaces

```
}
```

The weekly pay needs to be computed as specified in the problem description:

```
public class HourlyEmployee extends Employee
{
    . .
    public double weeklyPay(int hoursWorked)
    {
        double pay = hoursWorked * hourlyWage;
        if (hours_worked > 40)
        {
            // Add overtime
            pay = pay + ((hoursWorked - 40) * 0.5) * hourlyWage;
        }
        return pay;
    }
}

public class SalariedEmployee extends Employee
{
    . .
    public double weeklyPay(int hoursWorked)
    {
        final int WEEKS_PER_YEAR = 52;
        return annualSalary / WEEKS_PER_YEAR;
    }
}
```

In the case of the Manager, we need to call the version from the SalariedEmployee superclass:

```
public class Manager extends Employee
{
    . .
    public double weeklyPay(int hours)
    {
        return super.weeklyPay(hours) + weeklyBonus;
    }
}
```

Step 8 Construct objects of different subclasses and process them.

In our sample program, we populate an array of employees and compute the weekly salaries:

```
Employee[] staff = new Employee[3];
staff[0] = new HourlyEmployee("Morgan, Harry", 30);
staff[1] = new SalariedEmployee("Lin, Sally", 52000);
staff[2] = new Manager("Smith, Mary", 104000, 50);

Scanner in = new Scanner(System.in);
for (Employee e : staff)
{
    System.out.print("Hours worked by " + e.getName() + ": ");
    int hours = in.nextInt();
    System.out.println("Salary: " + e.weeklyPay(hours));
}
```

The complete code for this program is contained in the ch09/worked_example_1 directory of your source code.

WORKED EXAMPLE 9.2

Investigating Number Sequences



In this Worked Example, we investigate properties of number sequences. A number sequence can be a sequence of measurements, prices, random values, or mathematical values (such as the sequence of prime numbers). There are many interesting properties that can be investigated. For example, you can look for hidden patterns or test whether a sequence is truly random.

Problem Statement Investigate how the last digit of each value is distributed. For a given sequence of values, produce a chart such as

```
0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```



© Norebbo/iStockphoto.

In order to produce arbitrary sequences, we declare an interface type with a single method:

```
public interface Sequence
{
    int next();
}
```

The `LastDigitDistribution` class analyzes sequences. It keeps an array of ten counters. Its `process` method receives a `Sequence` object and the number of values to process and updates the counters:

```
public void process(Sequence seq, int valuesToProcess)
{
    for (int i = 1; i <= valuesToProcess; i++)
    {
        int value = seq.next();
        int lastDigit = value % 10;
        counters[lastDigit]++;
    }
}
```

Note that this method has no knowledge of how the sequence values are produced.

To analyze a specific sequence, you provide a class that implements the `Sequence` interface. Here are two examples: the sequence of perfect squares (0 1 4 9 16 25 ...) and a sequence of random integers.

```
public class SquareSequence implements Sequence
{
    private int n;

    public int next()
    {
        n++;
        return n * n;
    }
}
```

WE6 Chapter 9 Inheritance and Interfaces

```
}

public class RandomSequence implements Sequence
{
    public int next()
    {
        return (int) (Integer.MAX_VALUE * Math.random());
    }
}
```

The following class demonstrates the analysis process. Note the pattern of the last digits of the sequence of perfect squares.

```
public class SequenceDemo
{
    public static void main(String[] args)
    {
        LastDigitDistribution dist1 = new LastDigitDistribution();
        dist1.process(new SquareSequence(), 1000);
        dist1.display();
        System.out.println();

        LastDigitDistribution dist2 = new LastDigitDistribution();
        dist2.process(new RandomSequence(), 1000);
        dist2.display();
    }
}
```

Program Run

```
0: 100
1: 200
2: 0
3: 0
4: 200
5: 100
6: 200
7: 0
8: 0
9: 200

0: 105
1: 94
2: 81
3: 112
4: 89
5: 103
6: 103
7: 100
8: 108
9: 105
```

The complete program is contained in the `ch09/worked_example_2` directory of the book's companion code.

GRAPHICAL USER INTERFACES

CHAPTER GOALS

- To implement simple graphical user interfaces
- To add buttons, text fields, and other components to a frame window
- To handle events that are generated by buttons
- To write programs that display simple drawings



© Trout55/iStockphoto.

CHAPTER CONTENTS

10.1 FRAME WINDOWS 494

ST1 Adding the main Method to the Frame Class 498

10.2 EVENTS AND EVENT HANDLING 498

CE1 Modifying Parameter Types in the Implementing Method 506

CE2 Forgetting to Attach a Listener 506

PT1 Don't Use a Frame as a Listener 506

ST2 Local Inner Classes 507

ST3 Anonymous Inner Classes 508

J81 Lambda Expressions for Event Handling 509

10.3 PROCESSING TEXT INPUT 509

10.4 CREATING DRAWINGS 515

CE3 Forgetting to Repaint 524

CE4 By Default, Components Have Zero Width and Height 525

HT1 Drawing Graphical Shapes 525

WE1 Coding a Bar Chart Creator 

VE1 Solving Crossword Puzzles 



© Trout55/iStockphoto.

In this chapter, you will learn how to write graphical user-interface applications that contain buttons, text components, and graphical components such as charts. You will be able to process the events that are generated by button clicks, process the user input, and update the textual and graphical output.

10.1 Frame Windows

A graphical application shows information inside a **frame**: a window with a title bar, as shown in Figure 1. In the following sections, you will learn how to display a frame and how to place user-interface components inside it.

10.1.1 Displaying a Frame

To show a frame, construct a `JFrame` object, set its size, and make it visible.

To show a frame, carry out the following steps:

1. Construct an object of the `JFrame` class:

```
JFrame frame = new JFrame();
```

2. Set the size of the frame:

```
final int FRAME_WIDTH = 300;  
final int FRAME_HEIGHT = 400;  
frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
```

This frame will be 300 pixels wide and 400 pixels tall. If you omit this step the frame will be 0 by 0 pixels, and you won't be able to see it. (Pixels are the tiny dots from which digital images are composed.)



A graphical user interface is displayed inside a frame.

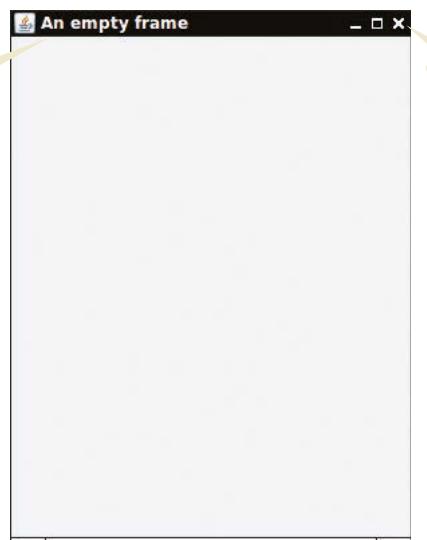


Figure 1 A Frame Window

3. If you'd like, set the title of the frame:

```
frame.setTitle("An empty frame");
```

If you omit this step, the title bar is simply left blank.

4. Set the “default close operation”:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

When the user closes the frame, the program automatically exits. Don't omit this step. If you do, the program keeps running even after the frame is closed.

5. Make the frame visible:

```
frame.setVisible(true);
```

The simple program below shows all of these steps. It produces the empty frame shown in Figure 1.

The `JFrame` class is a part of the `javax.swing` package. Swing is the nickname for a graphical user-interface toolkit that is part of the Java library. The “x” in `javax` denotes that Swing started out as a Java *extension* before it was added to the standard library.

sec01_01/EmptyFrameViewer.java

```

1 import javax.swing.JFrame;
2
3 /**
4  * This program displays an empty frame.
5 */
6 public class EmptyFrameViewer
7 {
8     public static void main(String[] args)
9     {
10        JFrame frame = new JFrame();
11
12        final int FRAME_WIDTH = 300;
13        final int FRAME_HEIGHT = 400;
14        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
15        frame.setTitle("An empty frame");
16        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17
18        frame.setVisible(true);
19    }
20}
```

10.1.2 Adding User-Interface Components to a Frame

An empty frame is not very interesting. You will want to add some user-interface components, such as buttons and text labels. However, if you add components directly to the frame, they get placed on top of each other.



© Eduardo Jose Bernardino/iStockphoto.

When building a graphical user interface, you add components to a frame.

Use a JPanel to group multiple user-interface components together.

If you have more than one component, put them into a **panel** (a container for other user-interface components), and then add the panel to the frame:

```
JPanel panel = new JPanel();
panel.add(button);
panel.add(label);
frame.add(panel);
```

You first construct the components, providing the text that should appear on them:

```
JButton button = new JButton("Click me!");
JLabel label = new JLabel("Hello, World!");
```

Then you add the components to the frame, as shown above. Figure 2 shows the result. When you run the program, you can click the button, but nothing will happen. You will see in Section 10.2.1 how to attach an action to a button.



Figure 2 A Frame with a Button and a Label

sec01_02/FilledFrameViewer.java

```
1 import javax.swing.JButton;
2 import javax.swing.JFrame;
3 import javax.swing.JLabel;
4 import javax.swing.JPanel;
5
6 /**
7  * This program shows a frame that is filled with two components.
8 */
9 public class FilledFrameViewer
{
10
11     public static void main(String[] args)
12     {
13         JFrame frame = new JFrame();
14
15         JButton button = new JButton("Click me!");
16         JLabel label = new JLabel("Hello, World!");
17
18         JPanel panel = new JPanel();
19         panel.add(button);
20         panel.add(label);
21         frame.add(panel);
22
23         final int FRAME_WIDTH = 300;
24         final int FRAME_HEIGHT = 100;
25         frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
26         frame.setTitle("A frame with two components");
27         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28
29         frame.setVisible(true);
30     }
31 }
```

10.1.3 Using Inheritance to Customize Frames

Declare a `JFrame` subclass for a complex frame.

As you add more user-interface components to a frame, the frame can get quite complex. Your programs will become easier to understand when you use inheritance for complex frames.

To do so, design a subclass of `JFrame`. Store the components as instance variables. Initialize them in the constructor of your subclass. This approach makes it easy to add helper methods for organizing your code.

It is also a good idea to set the frame size in the frame constructor. The frame usually has a better idea of the preferred size than the program displaying it.

For example,

```
public class FilledFrame extends JFrame
{
    // Use instance variables for components
    private JButton button;
    private JLabel label;

    private static final int FRAME_WIDTH = 300;
    private static final int FRAME_HEIGHT = 100;

    public FilledFrame()
    {
        // Now we can use a helper method
        createComponents();

        // It is a good idea to set the size in the frame constructor
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    private void createComponents()
    {
        button = new JButton("Click me!");
        label = new JLabel("Hello, World!");
        JPanel panel = new JPanel();
        panel.add(button);
        panel.add(label);
        add(panel);
    }
}
```

Of course, we still need a class with a `main` method:

```
public class FilledFrameViewer2
{
    public static void main(String[] args)
    {
        JFrame frame = new FilledFrame();
        frame.setTitle("A frame with two components");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



© TommL/Stockphoto.

In Java, you can use inheritance to customize a frame.



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download the complete `FilledFrame` program.



1. How do you display a square frame with a title bar that reads “Hello, World!”?
2. How can a program display two frames at once?
3. How can a program show a frame with two buttons labeled Yes and No?
4. Why does the `FilledFrameViewer2` class declare the `frame` variable to have class `JFrame`, not `FilledFrame`?
5. How many Java source files are required by the application in Section 10.1.3 when we use inheritance to declare the frame class?
6. Why does the `createComponents` method of `FilledFrame` call `add(panel)`, whereas the `main` method of `FilledFrameViewer` calls `frame.add(panel)`?

Practice It Now you can try these exercises at the end of the chapter: R10.1, R10.4, E10.1.

Special Topic 10.1



Adding the `main` Method to the Frame Class

Have another look at the `FilledFrame` and `FilledFrameViewer2` classes. Some programmers prefer to combine these two classes, by adding the `main` method to the frame class:

```
public class FilledFrame extends JFrame
{
    ...
    public static void main(String[] args)
    {
        JFrame frame = new FilledFrame();
        frame.setTitle("A frame with two components");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public FilledFrame()
    {
        createComponents();
        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }
    ...
}
```

This is a convenient shortcut that you will find in many programs, but it does not separate the responsibilities between the frame class and the program.

10.2 Events and Event Handling

In an application that interacts with the user through a console window, user input is under control of the program. The program asks the user for input in a specific order. For example, a program might ask the user to supply first a name, then a dollar amount. But the programs that you use every day on your computer don’t work like that. In a program with a modern **graphical user interface**, the *user* is in control. The user can use both the mouse and the keyboard and can manipulate many parts of the user interface in any desired order. For example, the user can enter information into text fields, pull down menus, click buttons, and drag scroll bars in any order. The program must react to the user commands in whatever order they arrive. Having to

deal with many possible inputs in random order is quite a bit harder than simply forcing the user to supply input in a fixed order.

In the following sections, you will learn how to write Java programs that can react to user-interface events.

10.2.1 Listening to Events

User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.

Whenever the user of a graphical program types characters or uses the mouse anywhere inside one of the windows of the program, the program receives a notification that an **event** has occurred. For example, whenever the mouse moves a tiny interval over a window, a “mouse move” event is generated. Clicking a button or selecting a menu item generates an “action” event.

Most programs don’t want to be flooded by irrelevant events. For example, when a button is clicked with the mouse, the mouse moves over the button, then the mouse button is pressed, and finally the button is released. Rather than receiving all these mouse events, a program can indicate that it only cares about button clicks, not about the underlying mouse events. On the other hand, if the mouse input is used for drawing shapes on a virtual canvas, a program needs to closely track mouse events.

Every program must indicate which events it needs to receive. It does that by installing **event listener** objects. These objects are instances of classes that you must provide. The methods of your event listener classes contain the instructions that you want to have executed when the events occur.

To install a listener, you need to know the **event source**. The event source is the user-interface component, such as a button, that generates a particular event. You add an event listener object to the appropriate event sources. Whenever the event occurs, the event source calls the appropriate methods of all attached event listeners.

This sounds somewhat abstract, so let’s run through an extremely simple program that prints a message whenever a button is clicked. Button listeners must belong to a class that implements the `ActionListener` interface:

```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```

This particular interface has a single method, `actionPerformed`. It is your job to supply a class whose `actionPerformed` method contains the instructions that you want executed whenever the button is clicked. Here is a very simple example of such a listener class:

sec02_01/ClickListener.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3
4 /**
5  * An action listener that prints a message.
6 */
```



© Seriy Tryapitsyn/iStockphoto

In an event-driven user interface, the program receives an event whenever the user manipulates an input component.

An event listener belongs to a class created by the application programmer. Its methods describe the actions to be taken when an event occurs.

Event sources report on events. When an event occurs, the event source notifies all event listeners.

```

6  */
7  public class ClickListener implements ActionListener
8  {
9      public void actionPerformed(ActionEvent event)
10     {
11         System.out.println("I was clicked.");
12     }
13 }

```

Attach an ActionListener to each button so that your program can react to button clicks.

We ignore the event parameter variable of the actionPerformed method—it contains additional details about the event, such as the time at which it occurred. Note that the event handling classes are defined in the `java.awt.event` package. (AWT is the Abstract Window Toolkit, the Java library for dealing with windows and events.)

Once the listener class has been declared, we need to construct an object of the class and add it to the button:

```

ActionListener listener = new ClickListener();
button.addActionListener(listener);

```

Whenever the button is clicked, the Java event handling library calls

```
listener.actionPerformed(event);
```

As a result, the message is printed.

You can test this program out by opening a console window, starting the `ButtonViewer1` program from that console window, clicking the button, and watching the messages in the console window (see Figure 3).



Figure 3 Implementing an Action Listener

sec02_01/ButtonFrame1.java

```

1 import java.awt.event.ActionListener;
2 import javax.swing.JButton;
3 import javax.swing.JFrame;
4 import javax.swing.JPanel;
5
6 /**
7     This frame demonstrates how to install an action listener.
8 */
9 public class ButtonFrame1 extends JFrame
10 {
11     private static final int FRAME_WIDTH = 100;
12     private static final int FRAME_HEIGHT = 60;
13
14     public ButtonFrame1()
15     {
16         createComponents();
17         setSize(FRAME_WIDTH, FRAME_HEIGHT);
18     }

```

```

19
20     private void createComponents()
21     {
22         JButton button = new JButton("Click me!");
23         JPanel panel = new JPanel();
24         panel.add(button);
25         add(panel);
26
27         ActionListener listener = new ClickListener();
28         button.addActionListener(listener);
29     }
30 }
```

sec02_01/ButtonViewer1.java

```

1 import javax.swing.JFrame;
2
3 /**
4  * This program demonstrates how to install an action listener.
5 */
6 public class ButtonViewer1
7 {
8     public static void main(String[] args)
9     {
10        JFrame frame = new ButtonFrame1();
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        frame.setVisible(true);
13    }
14 }
```

10.2.2 Using Inner Classes for Listeners

In the preceding section, you saw how to specify button actions. The code for the button action is placed into a listener class. It is common to implement listener classes as **inner classes** like this:

```

public class ButtonFrame2 extends JFrame
{
    . .
    // This inner class is declared inside the frame class
    class ClickListener implements ActionListener
    {
        . .
    }

    private void createComponents()
    {
        button = new JButton("Click me!");
        ActionListener listener = new ClickListener();
        button.addActionListener(listener);
        . .
    }
}
```

An inner class is simply a class that is declared inside another class.

There are two advantages to making a listener class into an inner class. First, listener classes tend to be very short. You can put the inner class close to where it is needed, without cluttering up the remainder of the project. Moreover, inner classes

© angelhell/iStockphoto.



An inner class is a class that is declared inside another class.

Methods of an inner class can access variables from the surrounding class.

have a very attractive feature: Their methods can access instance variables and methods of the surrounding class.

This feature is particularly useful when implementing event handlers. It allows the inner class to access variables without having to receive them as constructor or method arguments.

Let's look at an example. Instead of printing the message "I was clicked", we want to show it in a label. If we make the action listener into an inner class of the frame class, its actionPerformed method can access the label instance variable and call the setText method, which changes the label text.

```
public class ButtonFrame2 extends JFrame
{
    private JButton button;
    private JLabel label;
    .
    .
    class ClickListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            // Accesses label variable from surrounding class
            label.setText("I was clicked");
        }
    }
    .
}
```

Having the listener as a regular class is unattractive—the listener would need to be constructed with a reference to the label field (see Exercise E10.5).

sec02_02/ButtonFrame2.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 public class ButtonFrame2 extends JFrame
9 {
10     private JButton button;
11     private JLabel label;
12
13     private static final int FRAME_WIDTH = 300;
14     private static final int FRAME_HEIGHT = 100;
15
16     public ButtonFrame2()
17     {
18         createComponents();
19         setSize(FRAME_WIDTH, FRAME_HEIGHT);
20     }
21
22     /**
23      An action listener that changes the label text.
24     */
25     class ClickListener implements ActionListener
26     {
27         public void actionPerformed(ActionEvent event)
28     }
```

```

29         label.setText("I was clicked.");
30     }
31 }
32
33 private void createComponents()
34 {
35     button = new JButton("Click me!");
36     ActionListener listener = new ClickListener();
37     button.addActionListener(listener);
38
39     label = new JLabel("Hello, World!");
40
41     JPanel panel = new JPanel();
42     panel.add(button);
43     panel.add(label);
44     add(panel);
45 }
46 }
```

10.2.3 Application: Showing Growth of an Investment

In this section, we will build a practical application with a graphical user interface. A frame displays the amount of money in a bank account. Whenever the user clicks a button, 5 percent interest is added, and the new balance is displayed (see Figure 4).

We need a button and a label for the user interface. We also need to store the current balance:

```

public class InvestmentFrame extends JFrame
{
    private JButton button;
    private JLabel resultLabel;
    private double balance;

    private static final double INTEREST_RATE = 5;
    private static final double INITIAL_BALANCE = 1000;
    . .
}
```

We initialize the balance when the frame is constructed. Then we add the button and label to a panel, and the panel to the frame:

```

public InvestmentFrame()
{
    balance = INITIAL_BALANCE;

    createComponents();
    setSize(FRAME_WIDTH, FRAME_HEIGHT);
}
```



Figure 4 Clicking the Button Grows the Investment

Now we are ready for the hard part—the event listener that handles button clicks. As in the preceding section, it is necessary to declare a class that implements the `ActionListener` interface, and to place the button action into the `actionPerformed` method. Our listener class adds interest and displays the new balance:

```
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double interest = balance * INTEREST_RATE / 100;
        balance = balance + interest;
        resultLabel.setText("Balance: " + balance);
    }
}
```

We make this class an inner class so that it can access the `balance` and `resultLabel` instance variables.

Finally, we need to add an instance of the listener class to the button:

```
private void createComponents()
{
    button = new JButton("Add Interest");
    ActionListener listener = new AddInterestListener();
    button.addActionListener(listener);
    . . .
}
```

Here is the complete program. It demonstrates how to add multiple components to a frame, by using a panel, and how to implement listeners as inner classes.

sec02_03/InvestmentFrame.java

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7
8 public class InvestmentFrame extends JFrame
9 {
10     private JButton button;
11     private JLabel resultLabel;
12     private double balance;
13
14     private static final int FRAME_WIDTH = 300;
15     private static final int FRAME_HEIGHT = 100;
16
17     private static final double INTEREST_RATE = 5;
18     private static final double INITIAL_BALANCE = 1000;
19
20     public InvestmentFrame()
21     {
22         balance = INITIAL_BALANCE;
23
24         createComponents();
25         setSize(FRAME_WIDTH, FRAME_HEIGHT);
26     }
27 }
```

```

28     /**
29      * Adds interest to the balance and updates the display.
30     */
31     class AddInterestListener implements ActionListener
32     {
33         public void actionPerformed(ActionEvent event)
34         {
35             double interest = balance * INTEREST_RATE / 100;
36             balance = balance + interest;
37             resultLabel.setText("Balance: " + balance);
38         }
39     }
40
41     private void createComponents()
42     {
43         button = new JButton("Add Interest");
44         ActionListener listener = new AddInterestListener();
45         button.addActionListener(listener);
46
47         resultLabel = new JLabel("Balance: " + balance);
48
49         JPanel panel = new JPanel();
50         panel.add(button);
51         panel.add(resultLabel);
52         add(panel);
53     }
54 }
```

sec02_03/InvestmentViewer.java

```

1  import javax.swing.JFrame;
2
3  /**
4   * This program shows the growth of an investment.
5  */
6  public class InvestmentViewer
7  {
8      public static void main(String[] args)
9      {
10         JFrame frame = new InvestmentFrame();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setVisible(true);
13     }
14 }
```

SELF CHECK



7. Which objects are the event source and the event listener in the ButtonViewer program?
8. Why is it legal to assign a ClickListener object to a variable of type ActionListener?
9. When do you call the actionPerformed method?
10. Why would an inner class method want to access a variable from a surrounding scope?
11. How do you place the "Balance: . . ." message to the left of the "Add Interest" button in InvestmentFrame.java?

Practice It

Now you can try these exercises at the end of the chapter: R10.7, E10.2, E10.5.

Common Error 10.1**Modifying Parameter Types in the Implementing Method**

When you implement an interface, you must declare each method *exactly* as it is specified in the interface. Accidentally making small changes to the parameter variable types is a common error. Here is the classic example,

```
class MyListener implements ActionListener
{
    public void actionPerformed()
        // Oops . . . forgot ActionEvent parameter variable
    {
        . . .
    }
}
```

As far as the compiler is concerned, this class fails to provide the method

```
public void actionPerformed(ActionEvent event)
```

You have to read the error message carefully and pay attention to the parameter variable and return types to find your error.

Common Error 10.2**Forgetting to Attach a Listener**

If you run your program and find that your buttons seem to be dead, double-check that you attached the button listener. The same holds for other user-interface components. It is a surprisingly common error to program the listener class and the event handler action without actually attaching the listener to the event source.

Programming Tip 10.1**Don't Use a Frame as a Listener**

In this book, we use inner classes for event listeners. That approach works for many different event types. Once you master the technique, you don't have to think about it anymore. Many development environments automatically generate code with inner classes, so it is a good idea to be familiar with them.

However, some programmers bypass the event listener classes and turn a frame into a listener, like this:

```
public class InvestmentFrame extends JFrame
    implements ActionListener // This approach is not recommended
{
    . . .

    public InvestmentFrame()
    {
        button = new JButton("Add Interest");
        button.addActionListener(this);
        . . .
    }

    public void actionPerformed(ActionEvent event)
    {
        . . .
    }
}
```

Now the `actionPerformed` method is a part of the `InvestmentFrame` class rather than part of a separate listener class. The listener is installed as this.

We don't recommend this technique. If the viewer class contains two buttons that each generate action events, then the `actionPerformed` method must investigate the event source, which leads to code that is tedious and error-prone.

Special Topic 10.2



Local Inner Classes

An inner class can be declared completely inside a method. For example,

```
public static void main(String[] args)
{
    .
    .
    .
    class ClickListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            .
            .
            .
        }
    }

    JButton button = new JButton("Click me");
    button.addActionListener(new ClickListener());
    .
    .
}
```

This places the inner class exactly where you need it, next to the button.

The methods of a class that is defined inside a method can access the variables of the enclosing method. Prior to Java 8, it was necessary to declare those variables as `final`. For example,

```
public static void main(String[] args)
{
    final JLabel label = new JLabel("Hello, World!");
    .
    .
    .
    class ClickListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            label.setText("I was clicked");
            // Accesses label variable from enclosing method
        }
    }

    .
    .
}

button.addActionListener(new ClickListener());
```

As of Java 8, the variable must be *effectively final*. Such a variable must behave like a `final` variable (that is, stay unchanged after it has been initialized), but it need not be declared with the `final` modifier. The requirement to be final or effectively final sounds quite restrictive. However, it is usually not an issue if the variable is an object reference. Keep in mind that an object variable is `final` when the variable always refers to the same object. The state of the object can change, but the variable can't refer to a different object. For example, in our program, we never intended to have the `label` variable refer to multiple labels, so there was no harm in declaring it as `final`.

However, you can't change a numeric or Boolean local variable from an inner class. For example, the following would not work:

```
public static void main(String[] args)
{
    final double balance = INITIAL_BALANCE;
    .
    .
}
```

```

class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double interest = balance * (1 + INTEREST_RATE);
        balance = balance + interest;
        // Error: Can't modify a final numeric variable
    }
}
...
}

```

The remedy is to use an object instead:

```

public static void main(String[] args)
{
    final BankAccount account = new BankAccount();
    account.deposit(INITIAL_BALANCE);

    class AddInterestListener implements ActionListener
    {
        public void actionPerformed(ActionEvent event)
        {
            double interest = balance * (1 + INTEREST_RATE);
            account.deposit(interest);
            // Ok—we don't change the reference, just the object's state
        }
    }
}
...
}

```

Special Topic 10.3



Anonymous Inner Classes

An entity is anonymous if it does not have a name. In a program, something that is only used once doesn't usually need a name. For example, you can replace

```

String buttonLabel = "Add Interest";
JButton button = new JButton(buttonLabel);

```

with

```
JButton button = new JButton("Add Interest");
```

The string "Add Interest" is an anonymous object. Programmers like anonymous objects, because they don't have to go through the trouble of coming up with a name. If you have struggled with the decision whether to call a label `l`, `label`, or `buttonLabel`, you'll understand this sentiment.

Event listeners often give rise to a similar situation. You construct a single object of an event listener class. Afterward, the class is never used again. In Java, it is possible to declare an anonymous class if all you ever need is a single object of the class.

Here is an example:

```

button = new JButton("Add Interest");
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        double interest = balance * (1 + INTEREST_RATE);
        account.deposit(interest);
    }
});

```

This means: Define a class that implements the `ActionListener` interface with the given `actionPerformed` method. Construct an object of that class and pass it to the `addActionListener` method.

Many programmers like this style because it is so compact. Moreover, GUI builders in integrated development environments often generate code of this form.

Java 8 Note 10.1



Lambda Expressions for Event Handling

Java 8 Note 9.3 showed you how to use lambda expressions for instances of classes that implement “functional” interfaces; that is, interfaces with a single abstract method. This includes event handlers such as `ActionListener` objects.

For example, instead of declaring a `ClickListener` class and adding an instance as a listener to a button, you can simply add the listener as follows:

```
button.addActionListener(  
    (ActionEvent event) -> System.out.println("I was clicked."));
```

10.3 Processing Text Input

We continue our discussion with graphical user interfaces that accept text input. Of course, a graphical application can receive text input by calling the `ShowInputDialog` method of the `JOptionPane` class, but popping up a separate dialog box for each input is not a natural user interface. Most graphical programs collect text input through **text components** (see Figures 5 and 7). In the following two sections, you will learn how to add text components to a graphical application, and how to read what the user types into them.

10.3.1 Text Fields

Use a `JTextField` component for reading a single line of input. Place a `JLabel` next to each text field.

The `JTextField` class provides a text field for reading a single line of text. When you construct a text field, you need to supply the width—the approximate number of characters that you expect the user to type.

```
final int FIELD_WIDTH = 10;  
rateField = new JTextField(FIELD_WIDTH);
```

Users can type additional characters, but then a part of the contents of the field becomes invisible.

You will want to label each text field so that the user knows what to type into it. Construct a `JLabel` object for each label:

```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

You want to give the user an opportunity to enter all information into the text fields before processing it. Therefore, you should supply a button that the user can press to indicate that the input is ready for processing.



Figure 5
An Application
with a Text Field

When that button is clicked, its `actionPerformed` method should read the user input from each text field, using the `getText` method of the `JTextField` class. The `getText` method returns a `String` object. In our sample program, we turn the string into a number, using the `Double.parseDouble` method. After updating the account, we show the balance in another label.

```
class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double rate = Double.parseDouble(rateField.getText());
        double interest = balance * rate / 100;
        balance = balance + interest;
        resultLabel.setText("Balance: " + balance);
    }
}
```

The following application is a useful prototype for a graphical user-interface front end for arbitrary calculations. You can easily modify it for your own needs. Place input components into the frame. In the `actionPerformed` method, carry out the needed calculations. Display the result in a label.

[sec03_01/InvestmentFrame2.java](#)

```
1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JButton;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7 import javax.swing.JTextField;
8
9 /**
10      A frame that shows the growth of an investment with variable interest.
11 */
12 public class InvestmentFrame2 extends JFrame
13 {
14     private static final int FRAME_WIDTH = 450;
15     private static final int FRAME_HEIGHT = 100;
16
17     private static final double DEFAULT_RATE = 5;
18     private static final double INITIAL_BALANCE = 1000;
19
20     private JLabel rateLabel;
21     private JTextField rateField;
22     private JButton button;
23     private JLabel resultLabel;
24     private double balance;
25
26     public InvestmentFrame2()
27     {
28         balance = INITIAL_BALANCE;
29
30         resultLabel = new JLabel("Balance: " + balance);
31
32         createTextField();
33         createButton();
34         createPanel();
35     }
```

```

36         setSize(FRAME_WIDTH, FRAME_HEIGHT);
37     }
38
39     private void createTextField()
40     {
41         rateLabel = new JLabel("Interest Rate: ");
42
43         final int FIELD_WIDTH = 10;
44         rateField = new JTextField(FIELD_WIDTH);
45         rateField.setText("") + DEFAULT_RATE);
46     }
47
48     /**
49      * Adds interest to the balance and updates the display.
50     */
51     class AddInterestListener implements ActionListener
52     {
53         public void actionPerformed(ActionEvent event)
54         {
55             double rate = Double.parseDouble(rateField.getText());
56             double interest = balance * rate / 100;
57             balance = balance + interest;
58             resultLabel.setText("Balance: " + balance);
59         }
60     }
61
62     private void createButton()
63     {
64         button = new JButton("Add Interest");
65
66         ActionListener listener = new AddInterestListener();
67         button.addActionListener(listener);
68     }
69
70     private void createPanel()
71     {
72         JPanel panel = new JPanel();
73         panel.add(rateLabel);
74         panel.add(rateField);
75         panel.add(button);
76         panel.add(resultLabel);
77         add(panel);
78     }
79 }

```

10.3.2 Text Areas

Use a `JTextArea` to show multiple lines of text.

In the preceding section, you saw how to construct text fields. A text field holds a single line of text. To display multiple lines of text, use the `JTextArea` class.

You can use a text area for reading or displaying multi-line text.



© Kyoungil leon/Stockphoto.

When constructing a text area, you can specify the number of rows and columns:

```
final int ROWS = 10; // Lines of text
final int COLUMNS = 30; // Characters in each row
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

Use the `setText` method to set the text of a text field or text area. The `append` method adds text to the end of a text area. Use newline characters to separate lines, like this:

```
textArea.append(balance + "\n");
```

If you want to use a text field or text area for display purposes only, call the `setEditable` method like this

```
textArea.setEditable(false);
```

Now the user can no longer edit the contents of the field, but your program can still call `setText` and `append` to change it.

As shown in Figure 6, the `JTextField` and `JTextArea` classes are subclasses of the class `JTextComponent`. The methods `setText` and `setEditable` are declared in the `JTextComponent` class and inherited by `JTextField` and `JTextArea`. However, the `append` method is declared in the `JTextArea` class.

To add scroll bars to a text area, use a `JScrollPane`, like this:

```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
JScrollPane scrollPane = new JScrollPane(textArea);
```

Then add the scroll pane to the panel. Figure 7 shows the result.

You can add scroll bars to any component with a `JScrollPane`.

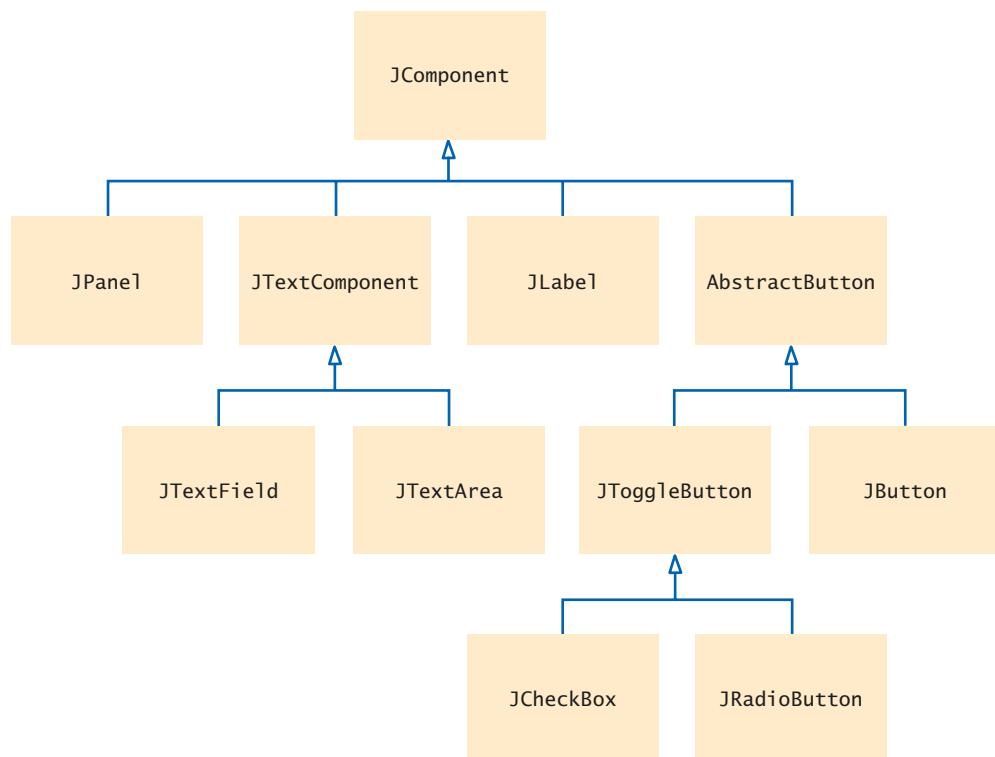


Figure 6 A Part of the Hierarchy of Swing User-Interface Components

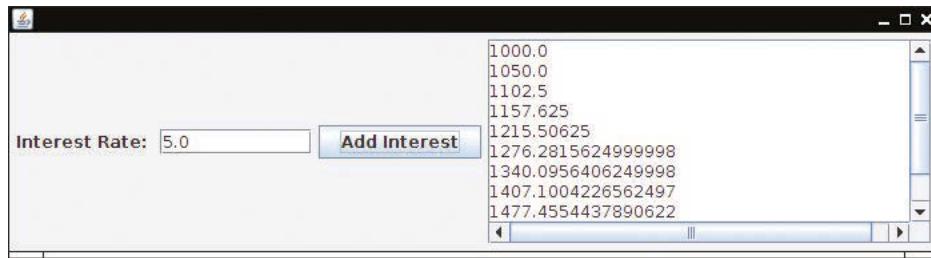


Figure 7 The Investment Application with a Text Area Inside Scroll Bars

The following sample program puts these concepts together. A user can enter numbers into the interest rate text field and then click on the “Add Interest” button. The interest rate is applied, and the updated balance is appended to the text area. The text area has scroll bars and is not editable.

This program is similar to the previous investment viewer program, but it keeps track of all the bank balances, not just the last one.

sec03_02/InvestmentFrame3.java

```

1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3  import javax.swing.JButton;
4  import javax.swing.JFrame;
5  import javax.swing.JLabel;
6  import javax.swing.JPanel;
7  import javax.swing.JScrollPane;
8  import javax.swing.JTextArea;
9  import javax.swing.JTextField;
10 
11 /**
12  * A frame that shows the growth of an investment with variable interest,
13  * using a text area.
14 */
15 public class InvestmentFrame3 extends JFrame
16 {
17     private static final int FRAME_WIDTH = 400;
18     private static final int FRAME_HEIGHT = 250;
19 
20     private static final int AREA_ROWS = 10;
21     private static final int AREA_COLUMNS = 30;
22 
23     private static final double DEFAULT_RATE = 5;
24     private static final double INITIAL_BALANCE = 1000;
25 
26     private JLabel rateLabel;
27     private JTextField rateField;
28     private JButton button;
29     private JTextArea resultArea;
30     private double balance;
31 
32     public InvestmentFrame3()
33     {
34         balance = INITIAL_BALANCE;
35         resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
36         resultArea.setText(balance + "\n");
37         resultArea.setEditable(false);

```

```

38
39     createTextField();
40     createButton();
41     createPanel();
42
43     setSize(FRAME_WIDTH, FRAME_HEIGHT);
44 }
45
46 private void createTextField()
47 {
48     rateLabel = new JLabel("Interest Rate: ");
49
50     final int FIELD_WIDTH = 10;
51     rateField = new JTextField(FIELD_WIDTH);
52     rateField.setText("") + DEFAULT_RATE);
53 }
54
55 class AddInterestListener implements ActionListener
56 {
57     public void actionPerformed(ActionEvent event)
58     {
59         double rate = Double.parseDouble(rateField.getText());
60         double interest = balance * rate / 100;
61         balance = balance + interest;
62         resultArea.append(balance + "\n");
63     }
64 }
65
66 private void createButton()
67 {
68     button = new JButton("Add Interest");
69
70     ActionListener listener = new AddInterestListener();
71     button.addActionListener(listener);
72 }
73
74 private void createPanel()
75 {
76     JPanel panel = new JPanel();
77     panel.add(rateLabel);
78     panel.add(rateField);
79     panel.add(button);
80     JScrollPane scrollPane = new JScrollPane(resultArea);
81     panel.add(scrollPane);
82     add(panel);
83 }
84 }

```

**SELF CHECK**

12. What happens if you omit the first `JLabel` object in the program of Section 10.3.1?
13. If a text field holds an integer, what expression do you use to read its contents?
14. What is the difference between a text field and a text area?
15. Why did the `InvestmentFrame3` program call `resultArea.setEditable(false)`?

- 16.** How would you modify the `InvestmentFrame3` program if you didn't want to use scroll bars?

Practice It Now you can try these exercises at the end of the chapter: R10.13, E10.9, E10.10.

10.4 Creating Drawings

You often want to include simple drawings such as graphs or charts in your programs. The Java library does not have any standard components for this purpose, but it is fairly easy to make your own drawings. The following sections show how.



© Alexey Avdeev/iStockphoto.

10.4.1 Drawing on a Component

We start out with a simple bar chart (see Figure 8) that is composed of three rectangles.

You cannot draw directly onto a frame. Instead, you add a component to the frame and draw on the component. To do so, extend the `JComponent` class and override its `paintComponent` method.

```
public class ChartComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Drawing instructions.
    }
}
```

In order to display a drawing, provide a class that extends the `JComponent` class.

Place drawing instructions inside the `paintComponent` method. That method is called whenever the component needs to be repainted.

When the component is shown for the first time, its `paintComponent` method is called automatically. The method is also called when the window is resized, or when it is shown again after it was hidden.

The `paintComponent` method receives an object of type `Graphics`. The `Graphics` object stores the graphics state—the current color, font, and so on, that are used for drawing operations. The `Graphics` class has methods for drawing geometric shapes. The call

```
g.fillRect(x, y, width, height)
```

draws a solid rectangle with upper-left corner (x, y) and the given width and height. If you call the `drawRect` method, you obtain the outline of the rectangle instead, without having the interior filled.

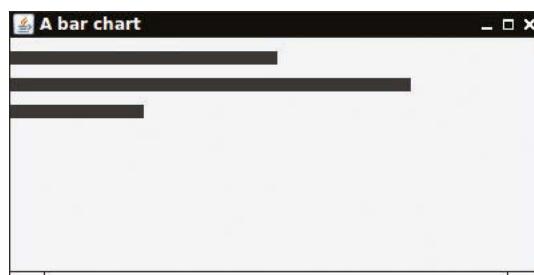


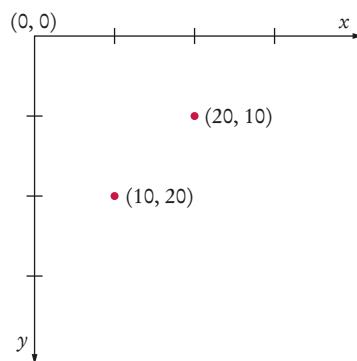
Figure 8 Drawing a Bar Chart

The `Graphics` class has methods to draw rectangles and other shapes.

Here we produce three filled rectangles. They line up on the left because they all have $x = 0$. They also all have the same height.

```
public class ChartComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        g.fillRect(0, 10, 200, 10);
        g.fillRect(0, 30, 300, 10);
        g.fillRect(0, 50, 100, 10);
    }
}
```

Note that the coordinate system is different from the one used in mathematics. The origin $(0, 0)$ is at the upper-left corner of the component, and the y -coordinate grows downward.



Here is the source code for the `ChartComponent` class. As you can see from the `import` statements, the `Graphics` class is part of the `java.awt` package.

sec04_01/ChartComponent.java

```
1 import java.awt.Graphics;
2 import javax.swing.JComponent;
3
4 /**
5  * A component that draws a bar chart.
6 */
7 public class ChartComponent extends JComponent
8 {
9     public void paintComponent(Graphics g)
10    {
11        g.fillRect(0, 10, 200, 10);
12        g.fillRect(0, 30, 300, 10);
13        g.fillRect(0, 50, 100, 10);
14    }
15 }
```

Now we need to add the component to a frame, and show the frame. Because the frame is so simple, we don't make a frame subclass. Here is the viewer class:

sec04_01/ChartViewer.java

```
1 import javax.swing.JComponent;
2 import javax.swing.JFrame;
```

```

3
4 public class ChartViewer
5 {
6     public static void main(String[] args)
7     {
8         JFrame frame = new JFrame();
9
10        frame.setSize(400, 200);
11        frame.setTitle("A bar chart");
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14        JComponent component = new ChartComponent();
15        frame.add(component);
16
17        frame.setVisible(true);
18    }
19 }
```

10.4.2 Ovals, Lines, Text, and Color

In the preceding section, you learned how to write a program that draws rectangles. Now we turn to additional graphical elements that allow you to draw quite a few interesting pictures.

To draw an oval, you specify its *bounding box* (see Figure 9) in the same way that you would specify a rectangle, namely by the x - and y -coordinates of the top-left corner and the width and height of the box. Then the call

```
g.drawOval(x, y, width, height);
```

draws the outline of an oval. To draw a circle, simply set the width and height to the same values:

```
g.drawOval(x, y, diameter, diameter);
```

Notice that (x, y) is the top-left corner of the bounding box, not the center of the circle.

If you want to fill the inside of an oval, use the `fillOval` method instead. Conversely, if you want only the outline of a rectangle, with no filling, use the `drawRect` method.

Use `drawRect`,
`drawOval`, and
`drawLine` to draw
geometric shapes.

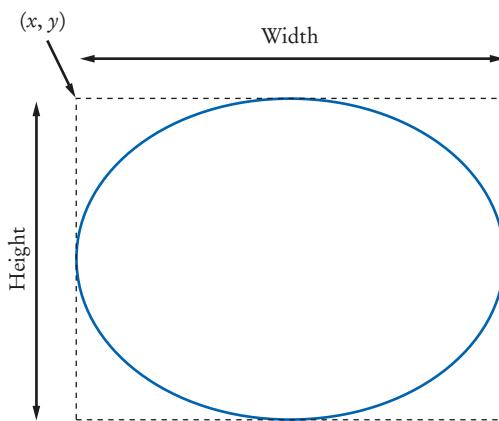


Figure 9 An Oval and Its Bounding Box

Figure 10
Basepoint and Baseline



The `drawString` method draws a string, starting at its basepoint.

To draw a line, call the `drawLine` method with the *x*- and *y*-coordinates of both end points:

```
g.drawLine(x1, y1, x2, y2);
```

You often want to put text inside a drawing, for example, to label some of the parts. Use the `drawString` method of the `Graphics` class to draw a string anywhere in a window. You must specify the string and the *x*- and *y*-coordinates of the basepoint of the first character in the string (see Figure 10). For example,

```
g.drawString("Message", 50, 100);
```

When you first start drawing, all shapes and strings are drawn with a black pen. To change the color, you need to supply an object of type `Color`. Java uses the RGB color model. That is, you specify a color by the amounts of the primary colors—red, green, and blue—that make up the color. The amounts are given as integers between 0 (primary color not present) and 255 (maximum amount present). For example,

```
Color magenta = new Color(255, 0, 255);
```

constructs a `Color` object with maximum red, no green, and maximum blue, yielding a bright purple color called magenta.

Table 1 Predefined Colors

Color	RGB Values
<code>Color.BLACK</code>	0, 0, 0
<code>Color.BLUE</code>	0, 0, 255
<code>Color.CYAN</code>	0, 255, 255
<code>Color.GRAY</code>	128, 128, 128
<code>Color.DARKGRAY</code>	64, 64, 64
<code>Color.LIGHTGRAY</code>	192, 192, 192
<code>Color.GREEN</code>	0, 255, 0
<code>Color.MAGENTA</code>	255, 0, 255
<code>Color.ORANGE</code>	255, 200, 0
<code>Color.PINK</code>	255, 175, 175
<code>Color.RED</code>	255, 0, 0
<code>Color.WHITE</code>	255, 255, 255
<code>Color.YELLOW</code>	255, 255, 0

When you set a new color in the graphics context, it is used for subsequent drawing operations.

For your convenience, a variety of colors have been predefined in the `Color` class. Table 1 shows those predefined colors and their RGB values. For example, `Color.PINK` has been predefined to be the same color as `new Color(255, 175, 175)`.

To draw a shape in a different color, first set the color of the `Graphics` object, then call the drawing method:

```
g.setColor(Color.YELLOW);
g.fillOval(350, 25, 35, 20); // Fills the oval in yellow
```

The following program puts all these shapes to work, creating a simple chart (see Figure 11).

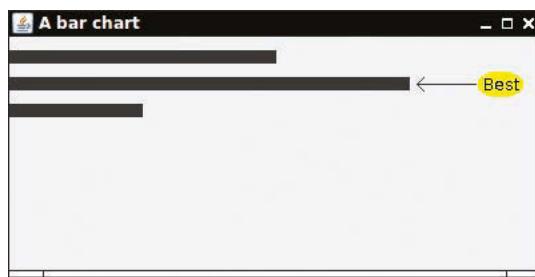


Figure 11 A Bar Chart with a Label

sec04_02/ChartComponent2.java

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3 import javax.swing.JComponent;
4
5 /**
6  * A component that draws a demo chart.
7 */
8 public class ChartComponent2 extends JComponent
9 {
10    public void paintComponent(Graphics g)
11    {
12        // Draw the bars
13        g.fillRect(0, 10, 200, 10);
14        g.fillRect(0, 30, 300, 10);
15        g.fillRect(0, 50, 100, 10);
16
17        // Draw the arrow
18        g.drawLine(350, 35, 305, 35);
19        g.drawLine(305, 35, 310, 30);
20        g.drawLine(305, 35, 310, 40);
21
22        // Draw the highlight and the text
23        g.setColor(Color.YELLOW);
24        g.fillOval(350, 25, 35, 20);
25        g.setColor(Color.BLACK);
26        g.drawString("Best", 355, 40);
27    }
28}
```

sec04_02/ChartViewer2.java

```

1 import javax.swing.JComponent;
2 import javax.swing.JFrame;
3
4 public class ChartViewer2
5 {
6     public static void main(String[] args)
7     {
8         JFrame frame = new JFrame();
9
10        frame.setSize(400, 200);
11        frame.setTitle("A bar chart");
12        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14        JComponent component = new ChartComponent2();
15        frame.add(component);
16
17        frame.setVisible(true);
18    }
19}

```

10.4.3 Application: Visualizing the Growth of an Investment

In this section, we will add a bar chart to the investment program of Section 10.3. Whenever the user clicks on the "Add Interest" button, another bar is added to the bar chart (see Figure 12).

The chart class of the preceding section produced a fixed bar chart. We will develop an improved version that can draw a chart with any values. The chart keeps an array list of the values:

```

public class ChartComponent extends JComponent
{
    private ArrayList<Double> values;
    private double maxValue;
    . .
}

```

When drawing the bars, we need to scale the values to fit into the chart. For example, if the investment program adds a value such as 10050 to the chart, we don't want to draw a bar that is 10,050 pixels long. In order to scale the values, we need to know the largest value that should still fit inside the chart. We will ask the user of the chart component to provide that maximum in the constructor:

```

public ChartComponent(double max)
{
    values = new ArrayList<Double>();
    maxValue = max;
}

```

We compute the width of a bar as

```
int barWidth = (int) (getWidth() * value / maxValue);
```

The `getWidth` method returns the width of the component in pixels. If the value to be drawn equals `maxValue`, the bar stretches across the entire component width.

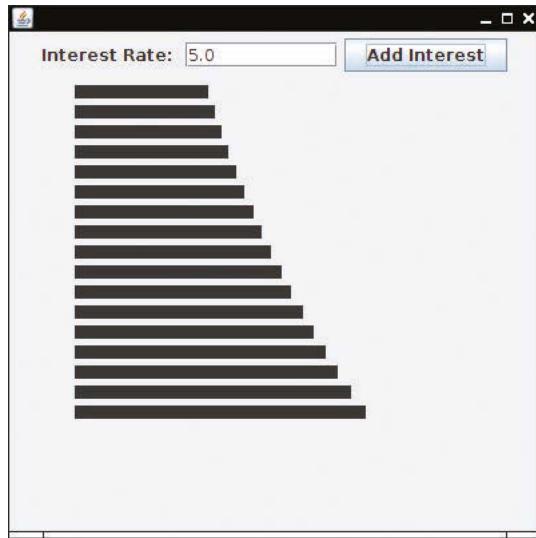


Figure 12 Clicking on the “Add Interest” Button Adds a Bar to the Chart

Here is the complete `paintComponent` method. We stack the bars horizontally and leave small gaps between them:

```
public void paintComponent(Graphics g)
{
    final int GAP = 5;
    final int BAR_HEIGHT = 10;

    int y = GAP;
    for (double value : values)
    {
        int barWidth = (int) (getWidth() * value / maxValue);
        g.fillRect(0, y, barWidth, BAR_HEIGHT);
        y = y + BAR_HEIGHT + GAP;
    }
}
```

Whenever the user clicks the “Add Interest” button, a value is added to the array list. Afterward, it is essential to call the `repaint` method:

```
public void append(double value)
{
    values.add(value);
    repaint();
}
```

The call to `repaint` forces a call to the `paintComponent` method. The `paintComponent` method redraws the component. Then the graph is drawn again, now showing the appended value.

Why not call `paintComponent` directly? The simple answer is that you can’t—you don’t have a `Graphics` object that you can pass as an argument. Instead, you need to ask the Swing library to make the call to `paintComponent` at its earliest convenience. That is what the `repaint` method does.

Call the `repaint` method whenever the state of a painted component changes.

When placing a painted component into a panel, you need to specify its preferred size.

We need to address another issue with painted components. If you place a painted component into a panel, you need to specify its preferred size. Otherwise, the panel will assume that the preferred size is 0 by 0 pixels, and you won't be able to see the component. Specifying the preferred size of a painted component is conceptually similar to specifying the number of rows and columns in a text area.

Call the `setPreferredSize` method with a `Dimension` object as argument. A `Dimension` argument wraps a width and a height into a single object. The call has the form

```
chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
```

That's all that is required to add a diagram to an application. Here is the code for the chart and frame classes; the viewer class is with the book's companion code.

sec04_03/ChartComponent.java

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3 import java.util.ArrayList;
4 import javax.swing.JComponent;
5
6 /**
7  * A component that draws a chart.
8 */
9 public class ChartComponent extends JComponent
10 {
11     private ArrayList<Double> values;
12     private double maxValue;
13
14     public ChartComponent(double max)
15     {
16         values = new ArrayList<Double>();
17         maxValue = max;
18     }
19
20     public void append(double value)
21     {
22         values.add(value);
23         repaint();
24     }
25
26     public void paintComponent(Graphics g)
27     {
28         final int GAP = 5;
29         final int BAR_HEIGHT = 10;
30
31         int y = GAP;
32         for (double value : values)
33         {
34             int barWidth = (int) (getWidth() * value / maxValue);
35             g.fillRect(0, y, barWidth, BAR_HEIGHT);
36             y = y + BAR_HEIGHT + GAP;
37         }
38     }
39 }
```

sec04_03/InvestmentFrame4.java

```

1 import java.awt.Dimension;
2 import java.awt.event.ActionEvent;
3 import java.awt.event.ActionListener;
```

```
4 import javax.swing.JButton;
5 import javax.swing.JFrame;
6 import javax.swing.JLabel;
7 import javax.swing.JPanel;
8 import javax.swing.JTextField;
9
10 /**
11  * A frame that shows the growth of an investment with variable interest,
12  * using a bar chart.
13 */
14 public class InvestmentFrame4 extends JFrame
15 {
16     private static final int FRAME_WIDTH = 400;
17     private static final int FRAME_HEIGHT = 400;
18
19     private static final int CHART_WIDTH = 300;
20     private static final int CHART_HEIGHT = 300;
21
22     private static final double DEFAULT_RATE = 5;
23     private static final double INITIAL_BALANCE = 1000;
24
25     private JLabel rateLabel;
26     private JTextField rateField;
27     private JButton button;
28     private ChartComponent chart;
29     private double balance;
30
31     public InvestmentFrame4()
32     {
33         balance = INITIAL_BALANCE;
34         chart = new ChartComponent(3 * INITIAL_BALANCE);
35         chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
36         chart.append(INITIAL_BALANCE);
37
38         createTextField();
39         createButton();
40         createPanel();
41
42         setSize(FRAME_WIDTH, FRAME_HEIGHT);
43     }
44
45     private void createTextField()
46     {
47         rateLabel = new JLabel("Interest Rate: ");
48
49         final int FIELD_WIDTH = 10;
50         rateField = new JTextField(FIELD_WIDTH);
51         rateField.setText("" + DEFAULT_RATE);
52     }
53
54     class AddInterestListener implements ActionListener
55     {
56         public void actionPerformed(ActionEvent event)
57         {
58             double rate = Double.parseDouble(rateField.getText());
59             double interest = balance * rate / 100;
60             balance = balance + interest;
61             chart.append(balance);
62         }
63     }
}
```

```

64
65     private void createButton()
66     {
67         button = new JButton("Add Interest");
68
69         ActionListener listener = new AddInterestListener();
70         button.addActionListener(listener);
71     }
72
73     private void createPanel()
74     {
75         JPanel panel = new JPanel();
76         panel.add(rateLabel);
77         panel.add(rateField);
78         panel.add(button);
79         panel.add(chart);
80         add(panel);
81     }
82 }
```

SELF CHECK

17. How do you modify the program in Section 10.4.1 to draw two squares?
18. What happens if you call `filloval` instead `fillRect` in the program of Section 10.4.1?
19. Give instructions to draw a circle with center (100, 100) and radius 25.
20. Give instructions to draw a letter “V” by drawing two line segments.
21. Give instructions to draw a string consisting of the letter “V”.
22. What are the RGB color values of `Color.BLUE`?
23. How do you draw a yellow square on a red background?
24. What would happen in the investment viewer program if we simply painted each bar as
`g.fillRect(0, y, value, BAR_HEIGHT);`
in the `paintComponent` method of the `ChartComponent` class?
25. What would happen if you omitted the call to `repaint` in the `append` method of the `ChartComponent` class?
26. What would happen if you omitted the call to `chart.setPreferredSize` in the `InvestmentFrame4` constructor?

Practice It Now you can try these exercises at the end of the chapter: R10.18, P10.1, P10.2.

Common Error 10.3**Forgetting to Repaint**

When you change the data in a painted component, the component is not automatically painted with the new data. You must call the `repaint` method of the component. Your component’s `paintComponent` method will then be invoked. Note that you should not call the `paintComponent` method directly.

The best place to call `repaint` is in the method of your component that modifies the data values:

```
void changeData( . . . )
{
    Update data values.
    repaint();
}
```

This is a concern only for your own painted components. When you make a change to a standard Swing component such as a `JLabel`, the component is automatically repainted.

Common Error 10.4



By Default, Components Have Zero Width and Height

You must be careful when you add a painted component, such as a component displaying a chart, to a panel. The default size for a `JComponent` is 0 by 0 pixels, and the component will not be visible. The remedy is to call the `setPreferredSize` method:

```
chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
```

This is an issue only for painted components. Buttons, labels, and so on, know how to compute their preferred size.

HOW TO 10.1



Drawing Graphical Shapes

Suppose you want to write a program that displays graphical shapes such as cars, aliens, charts, or any other images that can be obtained from rectangles, lines, and ellipses. These instructions give you a step-by-step procedure for decomposing a drawing into parts and implementing a program that produces the drawing.

Problem Statement Create a program to draw a national flag.

Step 1

Determine the shapes that you need for the drawing.

You can use the following shapes:

- Squares and rectangles
- Circles and ellipses
- Lines

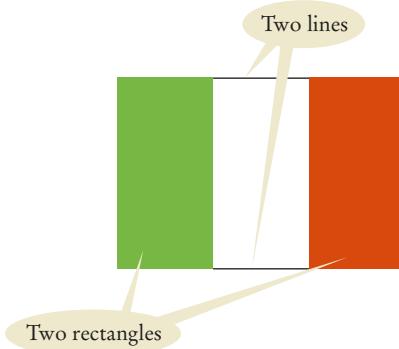
The outlines of these shapes can be drawn in any color, and you can fill the insides of these shapes with any color. You can also use text to label parts of your drawing.

Some national flag designs consist of three equally wide sections of different colors, side by side, as in the Italian flag shown below.



Punchstock.

You could draw such a flag using three rectangles. But if the middle rectangle is white, as it is, for example, in the flag of Italy (green, white, red), it is easier and looks better to draw a line on the top and bottom of the middle portion:



Step 2 Find the coordinates for the shapes.

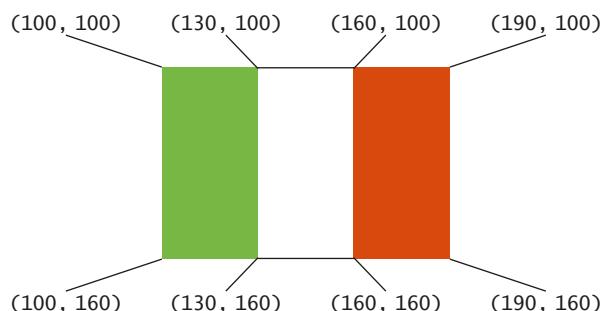
You now need to find the exact positions for the geometric shapes.

- For rectangles, you need the x - and y -position of the top-left corner, the width, and the height.
- For ellipses, you need the top-left corner, width, and height of the bounding rectangle.
- For lines, you need the x - and y -positions of the starting point and the end point.
- For text, you need the x - and y -position of the basepoint.

A commonly-used size for a window is 300 by 300 pixels. You may not want the flag crammed all the way to the top, so perhaps the upper-left corner of the flag should be at point (100, 100).

Many flags, such as the flag of Italy, have a width : height ratio of 3 : 2. (You can often find exact proportions for a particular flag by doing a bit of Internet research on one of several Flags of the World sites.) For example, if you make the flag 90 pixels wide, then it should be 60 pixels tall. (Why not make it 100 pixels wide? Then the height would be $100 \cdot 2 / 3 \approx 67$, which seems more awkward.)

Now you can compute the coordinates of all the important points of the shape:



Step 3 Write Java statements to draw the shapes.

In our example, there are two rectangles and two lines:

```

g.setColor(Color.GREEN);
g.fillRect(100, 100, 30, 60);

g.setColor(Color.RED);
g.fillRect(160, 100, 30, 60);

```

```

g.setColor(Color.BLACK);
g.drawLine(130, 100, 160, 100);
g.drawLine(130, 160, 160, 160);

```

If you are more ambitious, then you can express the coordinates in terms of a few variables. In the case of the flag, we have arbitrarily chosen the top-left corner and the width. All other coordinates follow from those choices. If you decide to follow the ambitious approach, then the rectangles and lines are determined as follows:

```

g.fillRect(xLeft, yTop, width / 3, width * 2 / 3);
.
.
g.fillRect(xLeft + 2 * width / 3, yTop, width / 3, width * 2 / 3);
.
.
g.drawLine(xLeft + width / 3, yTop, xLeft + width * 2 / 3, yTop);
g.drawLine(xLeft + width / 3, yTop + width * 2 / 3,
           xLeft + width * 2 / 3, yTop + width * 2 / 3);

```

Step 4

Consider using methods or classes for repetitive steps.

Do you need to draw more than one flag? Perhaps with different sizes? Then it is a good idea to design a method or class, so you won't have to repeat the same drawing instructions.

For example, you can write a method

```

void drawItalianFlag(Graphics g, int xLeft, int yTop, int width)
{
    Draw a flag at the given location and size.
}

```

Place the instructions from the preceding step into this method. Then you can call

```

drawItalianFlag(g, 10, 10, 100);
drawItalianFlag(g, 10, 125, 150);

```

in the `paintComponent` method to draw two flags.

Step 5

Place the drawing instructions in the `paintComponent` method.

```

public class ItalianFlagComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        Drawing instructions.
    }
}

```

If your drawing is simple, simply place all drawing statements here. Otherwise, call the methods you created in Step 4.

Step 6

Write the viewer class.

Provide a viewer class, with a `main` method in which you construct a frame, add your component, and make your frame visible. The viewer class is completely routine; you only need to change a single line to show a different component.

```

public class ItalianFlagViewer
{
    public static void main(String[] args)
    {
        JFrame frame = new JFrame();

        frame.setSize(300, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JComponent component = new ItalianFlagComponent();
        frame.add(component);
    }
}

```



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download the complete flag drawing program.

```

        frame.setVisible(true);
    }
}

```

**WORKED EXAMPLE 10.1****Coding a Bar Chart Creator**

In this Worked Example, we will develop a simple program for creating bar charts. The user enters labels and values for the bars, and the program displays the chart. Go to wiley.com/go/bj102examples and download Worked Example 10.1.

**VIDEO EXAMPLE 10.1****Solving Crossword Puzzles**

In this Video Example, we develop a program that finds words for solving a crossword puzzle. Go to wiley.com/go/bj102videos to view Video Example 10.1.



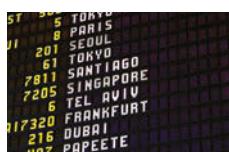
© paul jantz / iStockphoto.

CHAPTER SUMMARY**Display frames and add components inside frames.**

- To show a frame, construct a `JFrame` object, set its size, and make it visible.
- Use a `JPanel` to group multiple user-interface components together.
- Declare a `JFrame` subclass for a complex frame.

**Explain the event concept and handle button events.**

- User-interface events include key presses, mouse moves, button clicks, menu selections, and so on.
- An event listener belongs to a class created by the application programmer. Its methods describe the actions to be taken when an event occurs.
- Event sources report on events. When an event occurs, the event source notifies all event listeners.
- Attach an `ActionListener` to each button so that your program can react to button clicks.
- Methods of an inner class can access variables from the surrounding class.

**Use text components for reading text input.**

- Use a `JTextField` component for reading a single line of input. Place a `JLabel` next to each text field.
- Use a `JTextArea` to show multiple lines of text.
- You can add scroll bars to any component with a `JScrollPane`.

Create simple drawings with rectangles, ovals, lines, and text.

- In order to display a drawing, provide a class that extends the `JComponent` class.
- Place drawing instructions inside the `paintComponent` method. That method is called whenever the component needs to be repainted.
- The `Graphics` class has methods to draw rectangles and other shapes.
- Use `drawRect`, `drawOval`, and `drawLine` to draw geometric shapes.
- The `drawString` method draws a string, starting at its basepoint.
- When you set a new color in the graphics context, it is used for subsequent drawing operations.
- Call the `repaint` method whenever the state of a painted component changes.
- When placing a painted component into a panel, you need to specify its preferred size.



STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

<code>java.awt.Color</code>	<code>java.awt.Graphics</code>	<code>javax.swing.JFrame</code>
<code>java.awt.Component</code>	<code>setColor</code>	<code>setDefaultCloseOperation</code>
<code>getHeight</code>	<code>drawLine</code>	<code>javax.swing.JButton</code>
<code>getWidth</code>	<code>drawOval</code>	<code>javax.swing.JLabel</code>
<code>repaint</code>	<code>drawRect</code>	<code>javax.swing.JPanel</code>
<code>setPreferredSize</code>	<code>drawString</code>	<code>javax.swing.JScrollPane</code>
<code>setSize</code>	<code>fillOval</code>	<code>javax.swing.JTextArea</code>
<code>setVisible</code>	<code>fillRect</code>	<code>append</code>
<code>java.awt.Container</code>	<code>java.awt.event.ActionEvent</code>	<code>javax.swing.JTextField</code>
<code>add</code>	<code>java.awt.event.ActionListener</code>	<code>javax.swing.text.JTextComponent</code>
<code>java.awt.Dimension</code>	<code>actionPerformed</code>	<code>getText</code>
<code>java.awt.Frame</code>	<code>javax.swing.AbstractButton</code>	<code>isEditable</code>
<code>setTitle</code>	<code>addActionListener</code>	<code>setEditable</code>
	<code>javax.swing.JComponent</code>	<code>setText</code>
	<code>paintComponent</code>	

REVIEW EXERCISES

- **R10.1** What is the difference between a frame and a panel?
- **R10.2** From a programmer's perspective, what is the most important difference between the user interface of a console application and a graphical application?
- **R10.3** Why are separate viewer and frame classes used for graphical programs?
- **R10.4** What happens if you add a button and a label directly to a `JFrame` without using a `JPanel`? What happens if you add the label first? Try it out, by modifying the program in Section 10.1.2, and report your observations.
- **R10.5** What is an event object? An event source? An event listener?
- **R10.6** Who calls the `actionPerformed` method of an event listener? When does the call to the `actionPerformed` method occur?
- **R10.7** You can exit a graphical program by calling `System.exit(0)`. Describe how to provide an Exit button that functions in the same way as closing the window. Should you still call `setDefaultCloseOperation` on the frame?

- **R10.8** How would you add a counter to the program in Section 10.2.1 that prints how often the button has been clicked? Where is the counter updated?
- **R10.9** How would you add a counter to the program in Section 10.2.2 that shows how often the button has been clicked? Where is the counter updated? Where is it displayed?
- **R10.10** How would you reorganize the `InvestmentViewer` program in Section 10.2.3 if you needed to make `AddInterestListener` into a top-level class (that is, not an inner class)?
- **R10.11** Why are we using inner classes for event listeners? If Java did not have inner classes, could we still implement event listeners? How?
- **R10.12** Is it a requirement to use inheritance for frames, as described in Section 10.1.3? (*Hint:* Consider Special Topic 10.1.)
- **R10.13** What is the difference between a label, a text field, and a text area?
- **R10.14** Name a method that is declared in `JTextArea`, a method that `JTextArea` inherits from `JTextComponent`, and a method that `JTextArea` inherits from `JComponent`.
- **R10.15** Why did the program in Section 10.3.2 use a text area and not a label to show how the interest accumulates? How could you have achieved a similar effect with an array of labels?
- **R10.16** Who calls the `paintComponent` method of a component? When does the call to the `paintComponent` method occur?
- **R10.17** In the program of Section 10.4.2, why was the oval drawn before the string?
- **R10.18** How would you modify the chart component in Section 10.4.3 to draw a vertical bar chart? (*Careful:* The y -values grow downward.)
- **R10.19** How do you specify a text color?
- **R10.20** What is the difference between the `paintComponent` and `repaint` methods?
- **R10.21** Explain why the call to the `getWidth` method in the `ChartComponent` class has no explicit parameter.
- **R10.22** How would you modify the `drawItalianFlag` method in How To 10.1 to draw any flag with a white vertical stripe in the middle and two arbitrary colors to the left and right?

PRACTICE EXERCISES

- **E10.1** Write a program that shows a square frame filled with 100 buttons labeled 1 to 100. Nothing needs to happen when you press any of the buttons.
- **E10.2** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it prints a message “I was clicked n times!” whenever the button is clicked. The value n should be incremented with each click.
- **E10.3** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it has two buttons, each of which prints a message “I was clicked n times!” whenever the button is clicked. Each button should have a separate click count.

- **E10.4** Enhance the `ButtonViewer1` program in Section 10.2.1 so that it has two buttons labeled A and B, each of which prints a message “Button *x* was clicked!”, where *x* is A or B.
- **E10.5** Implement a `ButtonViewer1` program as in Exercise E10.3 using only a single listener class. *Hint:* Pass the button label to the constructor of the listener.
- **E10.6** Enhance the `ButtonViewer1` program so that it prints the date and time at which the button was clicked. *Hint:* `System.out.println(new java.util.Date())` prints the current date and time.
- **E10.7** Implement the `ClickListener` in the `ButtonViewer2` program of Section 10.2.2 as a regular class (that is, not an inner class). *Hint:* Store a reference to the label. Add a constructor to the listener class that sets the reference.
- **E10.8** Add error handling to the program in Section 10.3.2. If the interest rate is not a floating-point number, or if it less than 0, display an error message, using a JOptionPane (see Special Topic 2.5).
- **E10.9** Write a graphical application simulating a bank account. Supply text fields and buttons for depositing and withdrawing money, and for displaying the current balance in a label.
- **E10.10** Write a graphical application describing an earthquake, as in Section 3.3. Supply a text field and button for entering the strength of the earthquake. Display the earthquake description in a label.
- **E10.11** Write a graphical application for computing statistics of a data set. Supply a text field and button for adding floating-point values, and display the current minimum, maximum, and average in a label.
- **E10.12** Write an application with three labeled text fields, one each for the initial amount of a savings account, the annual interest rate, and the number of years. Add a button “Calculate” and a read-only text area to display the balance of the savings account after the end of each year.
- **E10.13** In the application from Exercise E10.12, replace the text area with a bar chart that shows the balance after the end of each year.
- **E10.14** Write a graphics program that draws your name in red, contained inside a blue rectangle. Provide a class `NameViewer` and a class `NameComponent`.
- **E10.15** Write a graphics program that draws 12 strings, one each for the 12 standard colors, besides `Color.WHITE`, each in its own color. Provide a class `ColorNameViewer` and a class `ColorNameComponent`.
- **E10.16** Write a program that draws two solid squares: one in pink and one in purple. Use a standard color for one of them and a custom color for the other. Provide a class `TwoSquareViewer` and a class `TwoSquareComponent`.
- **E10.17** Reimplement the program in Section 10.2.3, specifying the listener with a lambda expression (see Java 8 Note 10.1).



- E10.18** Reimplement the program in Section 10.3.1, specifying the listener with a lambda expression (see Java 8 Note 10.1).
- E10.19** Reimplement the program in Section 10.3.2, specifying the listener with a lambda expression (see Java 8 Note 10.1).

PROGRAMMING PROJECTS

- P10.1** Write a program to plot the following face. Provide a class FaceViewer and a class FaceComponent.



- P10.2** Draw a “bull’s eye”—a set of concentric rings in alternating black and white colors.
Hint: Fill a black circle, then fill a smaller white circle on top, and so on. Your program should be composed of classes BullsEyeComponent and BullsEyeViewer.



- P10.3** Write a program that draws a picture of a house. It could be as simple as the accompanying figure, or if you like, make it more elaborate (3-D, skyscraper, marble columns in the entryway, whatever).



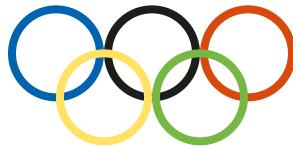
- P10.4** Extend Exercise P10.3 by supplying a drawHouse method in which you can specify the position and size. Then populate your frame with a few houses of different sizes.
- P10.5** Extend Exercise P10.4 so that you can make the houses appear in different colors. The color should be passed as an argument to the drawHouse method. Populate your frame with houses of different colors.
- P10.6** Improve the output quality of the investment application in Section 10.3.2. Format the numbers with two decimal digits, using the String.format method. Set the font of the text area to a fixed width font, using the call

```
textArea.setFont(new Font(Font.MONOSPACED, Font.PLAIN, 12));
```

- P10.7** Write a program that draws a 3D view of a cylinder.



- P10.8** Write a program to plot the string “HELLO”, using only lines and circles. Do not call `drawString`, and do not use `System.out`. Make classes `LetterH`, `LetterE`, `LetterL`, and `LetterO`.
- P10.9** Modify the `drawItalianFlag` method in How To 10.1 to draw any flag with three horizontal colored stripes. Write a program that displays the German and Hungarian flags.
- P10.10** Write a program that displays the Olympic rings. Color the rings in the Olympic colors. Provide a method `drawRing` that draws a ring of a given position and color.



- P10.11** Write a program that prompts the user to enter an integer in a text field. When a Draw button is clicked, draw as many rectangles at random positions in a component as the user requested.
- P10.12** Write a program that asks the user to enter an integer n into a text field. When a Draw button is clicked, draw an n -by- n grid in a component.
- P10.13** Write a program that has a Draw button and a component in which a random mixture of rectangles, ellipses, and lines, with random positions, is displayed each time the Draw button is clicked.
- P10.14** Make a bar chart to plot the following data set. Label each bar. Provide a class `BarChartViewer` and a class `BarChartComponent`.

Bridge Name	Longest Span (ft)
Golden Gate	4,200
Brooklyn	1,595
Delaware Memorial	2,150
Mackinac	3,800

- P10.15** Write a program that draws a clock face with a time that the user enters in two text fields (one for the hours, one for the minutes).
Hint: You need to determine the angles of the hour hand and the minute hand. The angle of the minute hand is easy; the minute hand travels 360 degrees in 60 minutes. The angle of the hour hand is harder; it travels 360 degrees in 12×60 minutes.
- P10.16** Write a program that fills the window with a large ellipse, with a black outline and filled with your favorite color. The ellipse should touch the window boundaries, even if the window is resized.
- Business P10.17** Implement a graphical application that simulates a cash register. Provide a text field for the item price and two buttons for adding the item to the sale, one for taxable items and one for nontaxable items. In a text area, display the register tape that lists all items (labeling the taxable items with a *), followed by the amount due. Provide another button for starting a new sale.

■■ Business P10.18 Write a graphical application to implement a currency converter between euros and U.S. dollars, and vice versa. Provide two text fields for the euro and dollar amounts. Between them, place two buttons labeled > and < for updating the field on the right or left. For this exercise, use a conversion rate of 1 euro = 1.42 U.S. dollars.

■■ Business P10.19 Write a graphical application that produces a restaurant bill. Provide buttons for ten popular dishes or drink items. (You decide on the items and their prices.) Provide text fields for entering less popular items and prices. In a text area, show the bill, including tax and a suggested tip.



© Juanmonino/iStockphoto.

ANSWERS TO SELF-CHECK QUESTIONS

1. Modify the `EmptyFrameViewer` program as follows:

```
final int FRAME_WIDTH = 300;
final int FRAME_HEIGHT = 300;
...
frame.setTitle("Hello, World!");
```

2. Construct two `JFrame` objects, set each of their sizes, and call `setVisible(true)` on each of them.

3. Add the following panel to the frame:

```
JButton button1 = new JButton("Yes");
JButton button2 = new JButton("No");
 JPanel panel = new JPanel();
 panel.add(button1);
 panel.add(button2);
```

4. There was no need to invoke any methods that are specific to `FilledFrame`. It is always a good idea to use the most general type when declaring a variable.

5. Two: `FilledFrameViewer2`, `FilledFrame`.

6. It's an instance method of `FilledFrame`, so the frame is the implicit parameter.

7. The `button` object is the event source. The `listener` object is the event listener.

8. The `ClickListener` class implements the `ActionListener` interface.

9. You don't. The Swing library calls the method when the button is clicked.

10. Direct access is simpler than the alternative—passing the variable as an argument to a constructor or method.

11. First add `label` to the `panel`, then add `button`.

12. Then the text field is not labeled, and the user will not know its purpose.

13. `Integer.parseInt(textField.getText())`

14. A text field holds a single line of text; a text area holds multiple lines.

15. The text area is intended to display the program output. It does not collect user input.

16. Don't construct a `JScrollPane` but add the `resultArea` object directly to the panel.

17. Here is one possible solution:

```
g.fillRect(0, 0, 50, 50);
g.fillRect(0, 100, 50, 50);
```

18. The program shows three very elongated ellipses instead of the rectangles.

19. `g.drawOval(75, 75, 50, 50);`

20. `g.drawLine(0, 0, 10, 30);
g.drawLine(10, 30, 20, 0);`

21. `g.drawString("V", 0, 30);`

22. `0, 0, 255`

23. First fill a big red square, then fill a small yellow square inside:

```
g.setColor(Color.RED);
g.fillRect(0, 0, 200, 200);
g.setColor(Color.YELLOW);
g.fillRect(50, 50, 100, 100);
```

24. All the bars would stretch all the way to the right of the component because they would be much longer than the component's width.

25. The chart would not be repainted when the user hits the "Add Interest" button.

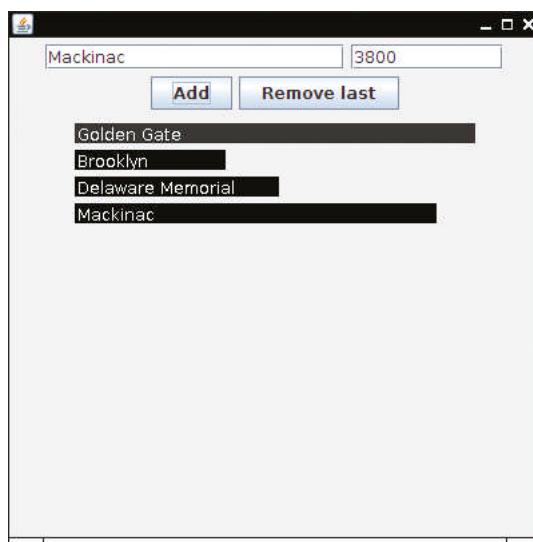
26. The chart would be shown at size 0 by 0; that is, it would be invisible.

WORKED EXAMPLE 10.1

Coding a Bar Chart Creator

Problem Statement We will develop a simple program for creating bar charts. The user enters labels and values for the bars, and the program displays the chart.

We also allow the user to fix mistakes by removing the last bar. Admittedly, the user interface is a bit limited. Worked Example 11.2 will improve on it, allowing users to use the mouse to edit the chart.



The Frame Class

Our program needs the following user-interface components:

- A text field for entering a bar label
- A text field for entering a bar value
- A button for adding a new bar with the given label and value
- A button for removing the last bar
- A component for drawing the chart

The chart component needs to provide two methods that support the button commands:

```
public void append(String label, double value)
public void removeLast()
```

For now, let us assume that those methods have been implemented. In the frame class, we need to provide action listeners that call them:

```
public class ChartFrame extends JFrame
{
    private JTextField labelField;
    private JTextField valueField;
    private JButton addButton;
    private JButton removeButton;
    private ChartComponent chart;

    class AddBarListener implements ActionListener
    {
```

WE2 Chapter 10 Graphical User Interfaces

```
public void actionPerformed(ActionEvent event)
{
    String label = labelField.getText();
    double value = Double.parseDouble(valueField.getText());
    chart.append(label, value);
}
}

class RemoveBarListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        chart.removeLast();
    }
}

private void createButtons()
{
    addButton = new JButton("Add");
    addButton.addActionListener(new AddBarListener());
    removeButton = new JButton("Remove last");
    removeButton.addActionListener(new RemoveBarListener());
}
.
.

}
```

The remainder of the frame class is straightforward. We need to construct the text field and the chart, place everything in a panel, and add the panel to the frame.

```
public class ChartFrame extends JFrame
{
    private static final int FRAME_WIDTH = 400;
    private static final int FRAME_HEIGHT = 400;

    private static final int CHART_WIDTH = 300;
    private static final int CHART_HEIGHT = 300;

    private static final String DEFAULT_LABEL = "Description";
    private static final double DEFAULT_VALUE = 100;

    .

    public ChartFrame()
    {
        chart = new ChartComponent();
        chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
        chart.append(DEFAULT_LABEL, DEFAULT_VALUE);

        createTextFields();
        createButtons();
        createPanel();

        setSize(FRAME_WIDTH, FRAME_HEIGHT);
    }

    private void createTextFields()
    {
        final int LABEL_FIELD_WIDTH = 20;
        labelField = new JTextField(LABEL_FIELD_WIDTH);
        labelField.setText(DEFAULT_LABEL);
        final int VALUE_FIELD_WIDTH = 10;
```

```

        valueField = new JTextField(VALUE_FIELD_WIDTH);
        valueField.setText("") + DEFAULT_VALUE);
    }

    . . .

    private void createPanel()
    {
        JPanel panel = new JPanel();
        panel.add(labelField);
        panel.add(valueField);
        panel.add(addButton);
        panel.add(removeButton);
        panel.add(chart);
        add(panel);
    }
}

```

Now let us turn to the bar chart itself. Unlike the chart of Section 10.4.3, this chart draws the labels in addition to the bars. A bar chart consists of multiple bars, each of which has a label and a value. It is best to make a class for a bar that holds both the label and the value. Then the chart component stores an array list of bars:

```

public class Bar
{
    private String label;
    private double value;

    public Bar(String aLabel, double aValue)
    {
        label = aLabel;
        value = aValue;
    }
    . . .
}

public class ChartComponent extends JComponent
{
    private ArrayList<Bar> bars;
    private double maxValue;
    . . .
}

```

To draw the chart, we ask that each bar draw itself. In general, this is a good strategy when you have an image that is made up of parts. Turn each part into an object with a `draw` method. A bar needs to know where it is situated in the chart—that is, the vertical displacement. It also needs to know how to scale itself in order to fit into the component. After all, the bar values are in units that make sense to the user (perhaps dollars or meters). Finally, in order to do any drawing, the `draw` method needs a `Graphics` object.

Here is the `draw` method for the `Bar` class:

```

public void draw(Graphics g, int y, double scale)
{
    final int GAP = 2;
    g.fillRect(0, y, (int) (value * scale), HEIGHT);
    g.setColor(Color.WHITE);
    g.drawString(label, GAP, y + HEIGHT - GAP);
    g.setColor(Color.BLACK);
}

```

Note that we draw the label in white on the black bar. When we are done, we need to restore the black color so that the next bar will be drawn in black.

WE4 Chapter 10 Graphical User Interfaces

When you look at this method, it becomes clearer why it is a good idea for the bar to draw itself, instead of placing the drawing instructions for all bars into the chart's `paintComponent` method. In this arrangement, the `ChartComponent` class need not to query the bar's data (that is, the label and width), and the `Bar` class doesn't need to provide methods that reveal them.

The `ChartComponent`'s `paintComponent` method simply asks each bar to draw itself. It's responsible for stacking the bars atop each other, and for fitting them horizontally inside the component.

```
public void paintComponent(Graphics g)
{
    final int GAP = 5;

    int y = GAP;
    double scale = getWidth() / maxValue;
    for (Bar b : bars)
    {
        b.draw(g, y, scale);
        y = y + Bar.HEIGHT + GAP;
    }
}
```

The `append` and `removeLast` methods add and remove a bar, then invoke `repaint` so that the changed chart is displayed.

```
public void append(String label, double value)
{
    bars.add(new Bar(label, value));
    if (value > maxValue) { maxValue = value; }
    repaint();
}

public void removeLast()
{
    int n = bars.size();
    if (n == 0) { return; }
    bars.remove(n - 1);
    repaint();
}
```

Finally, we need a routine viewer class that shows the frame. This completes the program.

It is worth reflecting on the division of labor between the component and frame classes. The `ChartComponent` class knows how to draw a chart, and it has methods for modifying the chart data. But it has no notion of a user interface. The same class could be used if we had a different user interface, perhaps with voice recognition instead of the text fields and buttons.

The `ChartFrame`, on the other hand, is all about the user interface. It handles the text fields and buttons. As soon as it knows what the user wants to do, it hands the work off to the chart component.

This is a useful division of labor, giving you guidance if the program needs to be enhanced. For a fancier rendering of the chart, improve the chart component. For more control over the chart's appearance, add user-interface components to the frame.

[worked_example_1/ChartViewer.java](#)

```
1 import javax.swing.JFrame;
2
3 /**
4      This program displays an editable bar chart.
5 */
6 public class ChartViewer
7 {
```

```

8   public static void main(String[] args)
9   {
10      JFrame frame = new ChartFrame();
11      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12      frame.setVisible(true);
13   }
14 }
```

worked_example_1/ChartFrame.java

```

1  import java.awt.Dimension;
2  import java.awt.event.ActionEvent;
3  import java.awt.event.ActionListener;
4  import javax.swing.JButton;
5  import javax.swing.JFrame;
6  import javax.swing.JLabel;
7  import javax.swing.JPanel;
8  import javax.swing.JTextField;
9
10 /**
11  * A frame that allows users to edit a bar chart.
12 */
13 public class ChartFrame extends JFrame
14 {
15     private static final int FRAME_WIDTH = 400;
16     private static final int FRAME_HEIGHT = 400;
17
18     private static final int CHART_WIDTH = 300;
19     private static final int CHART_HEIGHT = 300;
20
21     private static final String DEFAULT_LABEL = "Description";
22     private static final double DEFAULT_VALUE = 100;
23
24     private JTextField labelField;
25     private JTextField valueField;
26     private JButton addButton;
27     private JButton removeButton;
28     private ChartComponent chart;
29
30     public ChartFrame()
31     {
32         chart = new ChartComponent();
33         chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
34         chart.append(DEFAULT_LABEL, DEFAULT_VALUE);
35
36         createTextFields();
37         createButtons();
38         createPanel();
39
40         setSize(FRAME_WIDTH, FRAME_HEIGHT);
41     }
42
43     private void createTextFields()
44     {
45         final int LABEL_FIELD_WIDTH = 20;
46         labelField = new JTextField(LABEL_FIELD_WIDTH);
47         labelField.setText(DEFAULT_LABEL);
48         final int VALUE_FIELD_WIDTH = 10;
49         valueField = new JTextField(VALUE_FIELD_WIDTH);
50         valueField.setText("'" + DEFAULT_VALUE);
```

```

51     }
52
53     class AddBarListener implements ActionListener
54     {
55         public void actionPerformed(ActionEvent event)
56         {
57             String label = labelField.getText();
58             double value = Double.parseDouble(valueField.getText());
59             chart.append(label, value);
60         }
61     }
62
63     class RemoveBarListener implements ActionListener
64     {
65         public void actionPerformed(ActionEvent event)
66         {
67             chart.removeLast();
68         }
69     }
70
71     public void createButtons()
72     {
73         addButton = new JButton("Add");
74         addButton.addActionListener(new AddBarListener());
75         removeButton = new JButton("Remove last");
76         removeButton.addActionListener(new RemoveBarListener());
77     }
78
79     public void createPanel()
80     {
81         JPanel panel = new JPanel();
82         panel.add(labelField);
83         panel.add(valueField);
84         panel.add(addButton);
85         panel.add(removeButton);
86         panel.add(chart);
87         add(panel);
88     }
89 }

```

[worked_example_1/ChartComponent.java](#)

```

1  import java.awt.Color;
2  import java.awt.Graphics;
3  import java.util.ArrayList;
4  import javax.swing.JComponent;
5
6  /**
7   * A component that draws a chart.
8  */
9  public class ChartComponent extends JComponent
10 {
11     private ArrayList<Bar> bars;
12     private double maxValue;
13
14     /**
15      Constructs an empty chart.
16     */
17     public ChartComponent()
18     {

```

```

19     bars = new ArrayList<Bar>();
20     maxValue = 1;
21 }
22 /**
23  * Appends a bar to this chart.
24  * @param label the label for the bar
25  * @param value the value of the bar
26  */
27 public void append(String label, double value)
28 {
29     bars.add(new Bar(label, value));
30     if (value > maxValue) { maxValue = value; }
31     repaint();
32 }
33 /**
34  * Removes the last bar of this chart.
35  */
36 public void removeLast()
37 {
38     int n = bars.size();
39     if (n == 0) { return; }
40     bars.remove(n - 1);
41     repaint();
42 }
43 /**
44  * Paints this chart onto the specified Graphics object.
45  */
46 public void paintComponent(Graphics g)
47 {
48     final int GAP = 5;
49
50     int y = GAP;
51     double scale = getWidth() / maxValue;
52     for (Bar b : bars)
53     {
54         b.draw(g, y, scale);
55         y = y + Bar.HEIGHT + GAP;
56     }
57 }
58 }
```

worked_example_1/Bar.java

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3
4 /**
5  * A bar of a bar chart.
6  */
7 public class Bar
8 {
9     private String label;
10    private double value;
11
12    public static final int HEIGHT = 15;
13
14    /**
15     * Constructs a bar with a given label and value.
16     * @param label the label for this bar
17     * @param value the value of this bar
18 }
```

WE8 Chapter 10 Graphical User Interfaces

```
18 */  
19 public Bar(String aLabel, double aValue)  
20 {  
21     label = aLabel;  
22     value = aValue;  
23 }  
24  
25 /**  
26     Draws this bar.  
27     @param g the Graphics object  
28     @param y the top of the bar  
29     @param scale the horizontal scale factor for fitting the bar into the component  
30 */  
31 public void draw(Graphics g, int y, double scale)  
32 {  
33     final int GAP = 2;  
34     g.fillRect(0, y, (int) (value * scale), HEIGHT);  
35     g.setColor(Color.WHITE);  
36     g.drawString(label, GAP, y + HEIGHT - GAP);  
37     g.setColor(Color.BLACK);  
38 }  
39 }
```

ADVANCED USER INTERFACES

CHAPTER GOALS

- To use layout managers to arrange user-interface components in a container
- To become familiar with common user-interface components, such as radio buttons, check boxes, and menus
- To build programs that handle events generated by user-interface components
- To browse the Java documentation effectively



© Carlos Santa Maria/iStockphoto.

CHAPTER CONTENTS

11.1 LAYOUT MANAGEMENT 536

11.2 CHOICES 538

HT1 Laying Out a User Interface 546

PT1 Use a GUI Builder 548

WE1 Programming a Working Calculator

11.3 MENUS 549

11.4 EXPLORING THE SWING DOCUMENTATION 556

11.5 USING TIMER EVENTS FOR ANIMATIONS 561

11.6 MOUSE EVENTS 564

ST1 Keyboard Events 567

ST2 Event Adapters 568

WE2 Adding Mouse and Keyboard Support to the Bar Chart Creator

VE1 Designing a Baby Naming Program



© Carlos Santa Maria/iStockphoto.

The graphical applications with which you are familiar have many visual gadgets for information entry: buttons, scroll bars, menus, and so on. In this chapter, you will learn how to use the most common user-interface components in the Java Swing toolkit, and how to search the Java documentation for information about other components. You will also learn more about event handling, so you can use timer events in animations and process mouse events in interactive graphical programs.

11.1 Layout Management

User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.

Each container has a layout manager that directs the arrangement of its components.

Three useful layout managers are the border layout, flow layout, and grid layout.

When adding a component to a container with the border layout, specify the NORTH, SOUTH, WEST, EAST, or CENTER position.

Up to now, you have had limited control over the layout of user-interface components. You learned how to add components to a panel, and the panel arranged the components from left to right. However, in many applications, you need more sophisticated arrangements.

In Java, you build up user interfaces by adding components into containers such as **panels**. Each container has its own **layout manager**, which determines how components are laid out.

By default, a JPanel uses a **flow layout**. A flow layout simply arranges its components from left to right and starts a new row when there is no more room in the current row.

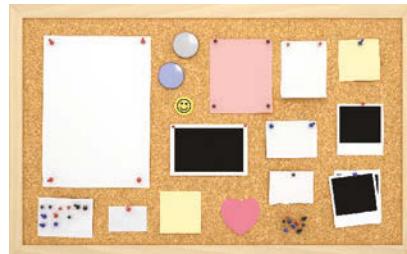
Another commonly used layout manager is the **border layout**. The border layout groups components into five areas: center, north, south, west, and east (see Figure 1). Each area can hold a single component, or it can be empty.

The border layout is the default layout manager for a frame (or, more technically, the frame's content pane). But you can also use the border layout in a panel:

```
panel.setLayout(new BorderLayout());
```

Now the panel is controlled by a border layout, not the flow layout. When adding a component, you specify the position, like this:

```
panel.add(component, BorderLayout.NORTH);
```



© Felix Mockel/iStockphoto.

A layout manager arranges user-interface components.

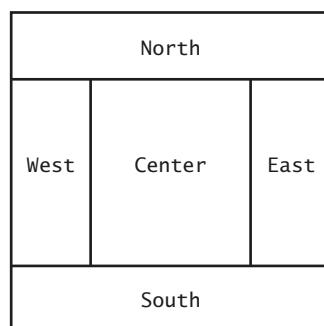


Figure 1
Components Expand to Fill Space in the Border Layout

7	8	9
4	5	6
1	2	3
0	.	CE

Figure 2 The Grid Layout

The content pane of a frame has a border layout by default. A panel has a flow layout by default.

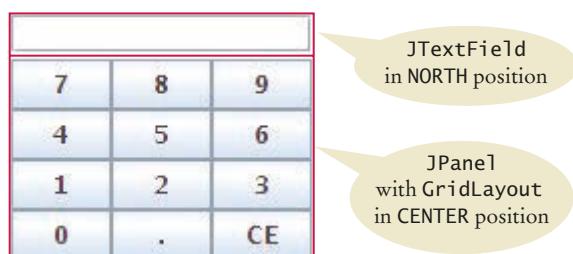
The **grid layout** manager arranges components in a grid with a fixed number of rows and columns. All components are resized so that they all have the same width and height. Like the border layout, it also expands each component to fill the entire allotted area. (If that is not desirable, you need to place each component inside a panel.) Figure 2 shows a number pad panel that uses a grid layout. To create a grid layout, you supply the number of rows and columns in the constructor, then add the components, row by row, left to right:

```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout(4, 3));
buttonPanel.add(button7);
buttonPanel.add(button8);
buttonPanel.add(button9);
buttonPanel.add(button4);
...

```

Sometimes you want to have a tabular arrangement of the components where columns have different sizes or one component spans multiple columns. A more complex layout manager called the *grid bag layout* can handle these situations. The grid bag layout is quite complex to use, however, and we do not cover it in this book. Another manager, the *group layout*, is designed for use by interactive tools—see Programming Tip 11.1 on page 548.

Fortunately, you can create acceptable-looking layouts in nearly all situations by nesting panels. You give each panel an appropriate layout manager. Panels don't have visible borders, so you can use as many panels as you need to organize your components. Figure 3 shows an example. The keypad buttons are contained in a panel with grid layout. That panel is itself contained in a larger panel with border layout. The text field is in the northern position of the larger panel.

**Figure 3** Nesting Panels

**FULL CODE EXAMPLE**

Go to wiley.com/go/bjlo2code to download the code for a calculator's user interface.

SELF CHECK

1. What happens if you place two buttons in the northern position of a border layout? Try it out with a small program.
2. How do you add two buttons to the northern position of a frame so that they are shown next to each other?
3. How can you stack three buttons one above the other?
4. What happens when you place one button in the northern position of a border layout and another in the center position? Try it out with a small program if you aren't sure.
5. Some calculators have a double-wide 0 button, as shown below. How can you achieve that?



Practice It Now you can try these exercises at the end of the chapter: R11.1, R11.3, E11.1.

11.2 Choices

In the following sections, you will see how to present a finite set of choices to the user. Which Swing component you use depends on whether the choices are mutually exclusive or not, and on the amount of space you have for displaying the choices.

11.2.1 Radio Buttons

For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.

If the choices are mutually exclusive, use a set of **radio buttons**. In a radio button set, only one button can be selected at a time. When the user selects another button in the same set, the previously selected button is automatically turned off. (These buttons are called radio buttons because they work like the station selector buttons on a car radio: If you select a new station,



© Michele Cornelius/
iStockphoto.

In an old fashioned radio, pushing down one station button released the others.

the old station is automatically deselected.) For example, in Figure 4, the font sizes are mutually exclusive. You can select small, medium, or large, but not a combination of them.

To create a set of radio buttons, first create each button individually, and then add all buttons in the set to a `ButtonGroup` object:

```
JRadioButton smallButton = new JRadioButton("Small");
JRadioButton mediumButton = new JRadioButton("Medium");
JRadioButton largeButton = new JRadioButton("Large");

ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
group.add(largeButton);
```

Note that the button group does *not* place the buttons close to each other in the container. The purpose of the button group is simply to find out which buttons to turn off when one of them is turned on. It is still your job to arrange the buttons on the screen.

The `isSelected` method is called to find out whether a button is currently selected or not. For example,

```
if (largeButton.isSelected()) { size = LARGE_SIZE; }
```

Unfortunately, there is no convenient way of finding out which button in a group is currently selected. You have to call `isSelected` on each button. Because users will expect one radio button in a radio button group to be selected, call `setSelected(true)` on the default radio button before making the enclosing frame visible.

If you have multiple button groups, it is a good idea to group them together visually. It is a good idea to use a panel for each set of radio buttons, but the panels themselves are invisible. You can add a *border* to a panel to make it visible. In Figure 4, for example, the panels containing the Size radio buttons and Style check boxes have borders.

You can place a border around a panel to group its contents visually.

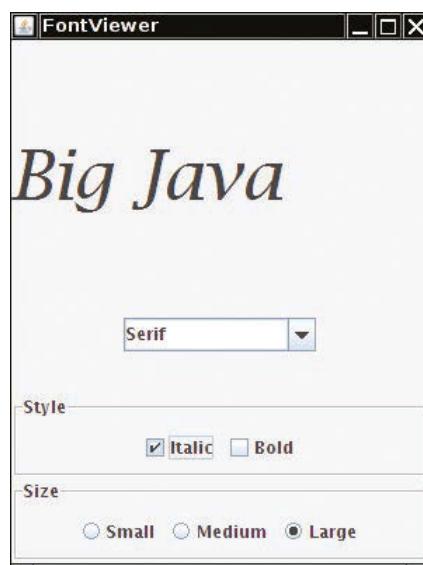


Figure 4 A Combo Box, Check Boxes, and Radio Buttons

There are a large number of border types. We will show only a couple of variations and leave it to the border enthusiasts to look up the others in the Swing documentation. The `EtchedBorder` class yields a border with a three-dimensional, etched effect. You can add a border to any component, but most commonly you apply it to a panel:

```
JPanel panel = new JPanel();
panel.setBorder(new EtchedBorder());
```

If you want to add a title to the border (as in Figure 4), you need to construct a `TitledBorder`. You make a titled border by supplying a basic border and then the title you want. Here is a typical example:

```
panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
```

11.2.2 Check Boxes

For a binary choice, use a check box.

A **check box** is a user-interface component with two states: checked and unchecked. You use a group of check boxes when one selection does not exclude another. For example, the choices for “Bold” and “Italic” in Figure 4 are not exclusive. You can choose either, both, or neither. Therefore, they are implemented as a set of separate check boxes. Radio buttons and check boxes have different visual appearances. Radio buttons are round and have a black dot when selected. Check boxes are square and have a check mark when selected.

You construct a check box by providing the name in the constructor:

```
JCheckBox italicCheckBox = new JCheckBox("Italic");
```

Because check box settings do not exclude each other, you do not place a set of check boxes inside a button group.

As with radio buttons, you use the `isSelected` method to find out whether a check box is currently checked or not.

11.2.3 Combo Boxes

For a large set of choices, use a combo box.

If you have a large number of choices, you don’t want to make a set of radio buttons, because that would take up a lot of space. Instead, you can use a **combo box**. This component is called a combo box because it is a combination of a list and a text field. The text field displays the name of the current selection. When you click on the arrow to the right of the text field of a combo box, a list of selections drops down, and you can choose one of the items in the list (see Figure 5).



Figure 5 An Open Combo Box

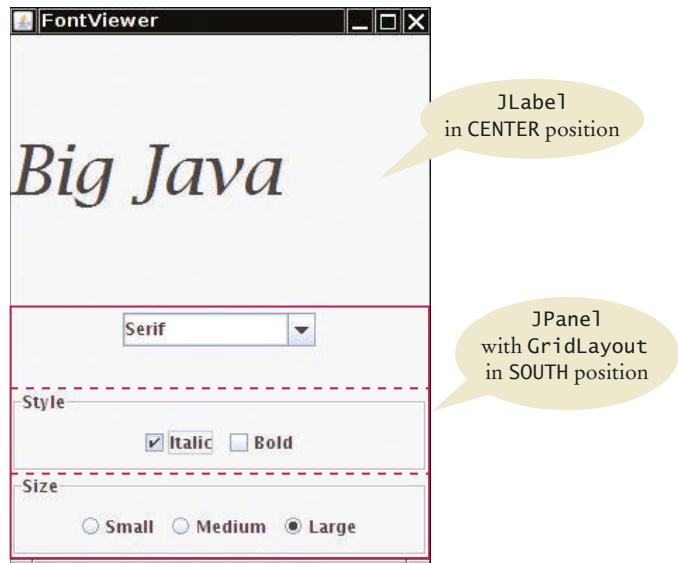


Figure 6 The Components of the FontFrame

If the combo box is *editable*, you can also type in your own selection. To make a combo box editable, call the `setEditable` method.

You add strings to a combo box with the `addItem` method.

```
JComboBox facenameCombo = new JComboBox();
facenameCombo.addItem("Serif");
facenameCombo.addItem("SansSerif");
. . .
```

You get the item that the user has selected by calling the `getSelectedItem` method. However, because combo boxes can store other objects in addition to strings, the `getSelectedItem` method has return type `Object`. Hence, in our example, you must cast the returned value back to `String`:

```
String selectedString = (String) facenameCombo.getSelectedItem();
```

You can select an item for the user with the `setSelectedItem` method.

Radio buttons, check boxes, and combo boxes generate an `ActionEvent` whenever the user selects an item. In the following program, we don't care which component was clicked—all components notify the same listener object. Whenever the user clicks on any one of them, we simply ask each component for its current content, using the `isSelected` and `getSelectedItem` methods. We then redraw the label with the new font.

Figure 6 shows how the components are arranged in the frame.

Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

sec02/FontViewer.java

```
1 import javax.swing.JFrame;
2
3 /**
4  * This program allows the user to view font effects.
5 */
6 public class FontViewer
7 {
```

```

8  public static void main(String[] args)
9  {
10    JFrame frame = new FontFrame();
11    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12    frame.setTitle("FontViewer");
13    frame.setVisible(true);
14  }
15 }
```

sec02/FontFrame.java

```

1  import java.awt.BorderLayout;
2  import java.awt.Font;
3  import java.awt.GridLayout;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import javax.swing.ButtonGroup;
7  import javax.swing.JButton;
8  import javax.swing.JCheckBox;
9  import javax.swing.JComboBox;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JPanel;
13 import javax.swing.JRadioButton;
14 import javax.swing.border.EtchedBorder;
15 import javax.swing.border.TitledBorder;
16
17 /**
18   This frame contains a text sample and a control panel
19   to change the font of the text.
20 */
21 public class FontFrame extends JFrame
22 {
23   private static final int FRAME_WIDTH = 300;
24   private static final int FRAME_HEIGHT = 400;
25
26   private JLabel label;
27   private JCheckBox italicCheckBox;
28   private JCheckBox boldCheckBox;
29   private JRadioButton smallButton;
30   private JRadioButton mediumButton;
31   private JRadioButton largeButton;
32   private JComboBox facenameCombo;
33   private ActionListener listener;
34
35 /**
36   Constructs the frame.
37 */
38 public FontFrame()
39 {
40   // Construct text sample
41   label = new JLabel("Big Java");
42   add(label, BorderLayout.CENTER);
43
44   // This listener is shared among all components
45   listener = new ChoiceListener();
46
47   createControlPanel();
48   setLabelFont();
```

```
49         setSize(FRAME_WIDTH, FRAME_HEIGHT);
50     }
51
52     class ChoiceListener implements ActionListener
53     {
54         public void actionPerformed(ActionEvent event)
55         {
56             setLabelFont();
57         }
58     }
59
60     /**
61      * Creates the control panel to change the font.
62     */
63     public void createControlPanel()
64     {
65         JPanel facenamePanel = createComboBox();
66         JPanel sizeGroupPanel = createCheckboxes();
67         JPanel styleGroupPanel = createRadioButtons();
68
69         // Line up component panels
70
71         JPanel controlPanel = new JPanel();
72         controlPanel.setLayout(new GridLayout(3, 1));
73         controlPanel.add(facenamePanel);
74         controlPanel.add(sizeGroupPanel);
75         controlPanel.add(styleGroupPanel);
76
77         // Add panels to content pane
78
79         add(controlPanel, BorderLayout.SOUTH);
80     }
81
82     /**
83      * Creates the combo box with the font style choices.
84      * @return the panel containing the combo box
85     */
86     public JPanel createComboBox()
87     {
88         facenameCombo = new JComboBox();
89         facenameCombo.addItem("Serif");
90         facenameCombo.addItem("SansSerif");
91         facenameCombo.addItem("Monospaced");
92         facenameCombo.setEditable(true);
93         facenameCombo.addActionListener(listener);
94
95         JPanel panel = new JPanel();
96         panel.add(facenameCombo);
97         return panel;
98     }
99
100    /**
101     * Creates the check boxes for selecting bold and italic styles.
102     * @return the panel containing the check boxes
103    */
104    public JPanel createCheckboxes()
105    {
106        italicCheckBox = new JCheckBox("Italic");
107        italicCheckBox.addActionListener(listener);
108    }
```

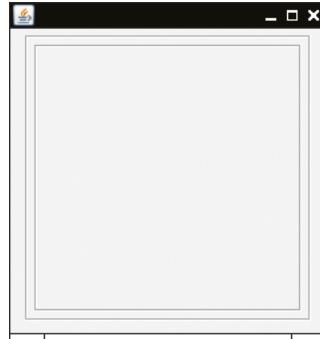
```
109     boldCheckBox = new JCheckBox("Bold");
110     boldCheckBox.addActionListener(listener);
111
112     JPanel panel = new JPanel();
113     panel.add(italicCheckBox);
114     panel.add(boldCheckBox);
115     panel.setBorder(new TitledBorder(new EtchedBorder(), "Style"));
116
117     return panel;
118 }
119
120 /**
121  * Creates the radio buttons to select the font size.
122  * @return the panel containing the radio buttons
123 */
124 JPanel createRadioButtons()
125 {
126     smallButton = new JRadioButton("Small");
127     smallButton.addActionListener(listener);
128
129     mediumButton = new JRadioButton("Medium");
130     mediumButton.addActionListener(listener);
131
132     largeButton = new JRadioButton("Large");
133     largeButton.addActionListener(listener);
134     largeButton.setSelected(true);
135
136     // Add radio buttons to button group
137
138     ButtonGroup group = new ButtonGroup();
139     group.add(smallButton);
140     group.add(mediumButton);
141     group.add(largeButton);
142
143     JPanel panel = new JPanel();
144     panel.add(smallButton);
145     panel.add(mediumButton);
146     panel.add(largeButton);
147     panel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
148
149     return panel;
150 }
151
152 /**
153  * Gets user choice for font name, style, and size
154  * and sets the font of the text sample.
155 */
156 public void setLabelFont()
157 {
158     // Get font name
159     String facename = (String) facenameCombo.getSelectedItem();
160
161     // Get font style
162
163     int style = 0;
164     if (italicCheckBox.isSelected())
165     {
166         style = style + Font.ITALIC;
167     }
```

```

168 if (boldCheckBox.isSelected())
169 {
170     style = style + Font.BOLD;
171 }
172
173 // Get font size
174
175 int size = 0;
176
177 final int SMALL_SIZE = 24;
178 final int MEDIUM_SIZE = 36;
179 final int LARGE_SIZE = 48;
180
181 if (smallButton.isSelected()) { size = SMALL_SIZE; }
182 else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }
183 else if (largeButton.isSelected()) { size = LARGE_SIZE; }
184
185 // Set font of text field
186
187 label.setFont(new Font(facename, style, size));
188 label.repaint();
189 }
190 }
```

SELF CHECK

6. What is the advantage of a JComboBox over a set of radio buttons? What is the disadvantage?
7. What happens when you put two check boxes into a button group? Try it out if you are not sure.
8. How can you nest two etched borders, like this?



9. Why do all user-interface components in the FontFrame class share the same listener?
10. Why was the combo box placed inside a panel? What would have happened if it had been added directly to the control panel?
11. How could the following user interface be improved?

Bold Yes No

Practice It Now you can try these exercises at the end of the chapter: R11.11, E11.3, E11.4.

HOW TO 11.1**Laying Out a User Interface**

A graphical user interface is made up of components such as buttons and text fields. The Swing library uses containers and layout managers to arrange these components. This How To explains how to group components into containers and how to pick the right layout managers.

Step 1 Make a sketch of your desired component layout.

Draw all the buttons, labels, text fields, and borders on a sheet of paper. Graph paper works best.

Here is an example—a user interface for ordering pizza. The user interface contains

- Three radio buttons
- Two check boxes
- A label: “Your Price:”
- A text field
- A border

Size	<input checked="" type="radio"/> Small	<input checked="" type="checkbox"/> Pepperoni
	<input type="radio"/> Medium	<input checked="" type="checkbox"/> Anchovies
	<input type="radio"/> Large	
Your Price:	<input type="text"/>	

Step 2 Find groupings of adjacent components with the same layout.

Usually, the component arrangement is complex enough that you need to use several panels, each with its own layout manager. Start by looking at adjacent components that are arranged top to bottom or left to right. If several components are surrounded by a border, they should be grouped together.

Here are the groupings from the pizza user interface:

Size	<input checked="" type="radio"/> Small	<input checked="" type="checkbox"/> Pepperoni
	<input type="radio"/> Medium	<input checked="" type="checkbox"/> Anchovies
	<input type="radio"/> Large	
Your Price:	<input type="text"/>	

Step 3 Identify layouts for each group.

When components are arranged horizontally, choose a flow layout. When components are arranged vertically, use a grid layout with one column.

In the pizza user interface example, you would choose

- A (3, 1) grid layout for the radio buttons
- A (2, 1) grid layout for the check boxes
- A flow layout for the label and text field

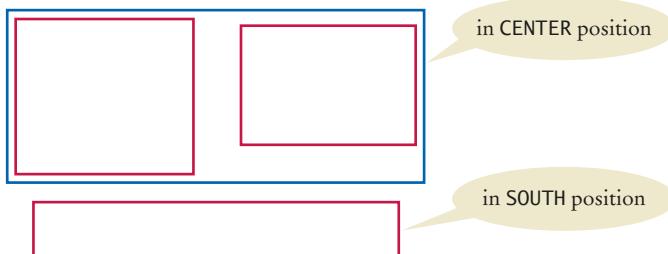
Step 4 Group the groups together.

Look at each group as one blob, and group the blobs together into larger groups, just as you grouped the components in the preceding step. If you note one large blob surrounded by smaller blobs, you can group them together in a border layout.

You may have to repeat the grouping again if you have a very complex user interface. You are done if you have arranged all groups in a single container.

For example, the three component groups of the pizza user interface can be arranged as:

- A group containing the first two component groups, placed in the center of a container with a border layout.
- The third component group, in the southern area of that container.



In this step, you may run into a couple of complications. The group “blobs” tend to vary in size more than the individual components. If you place them inside a grid layout, the grid layout forces them all to be the same size. Also, you occasionally would like a component from one group to line up with a component from another group, but there is no way for you to communicate that intent to the layout managers.

These problems can be overcome by using more sophisticated layout managers or implementing a custom layout manager. However, those techniques are beyond the scope of this book. Sometimes, you may want to start over with Step 1, using a component layout that is easier to manage. Or you can decide to live with minor imperfections of the layout. Don’t worry about achieving the perfect layout—after all, you are learning programming, not user-interface design.

Step 5

Write the code to generate the layout.

This step is straightforward but potentially tedious, especially if you have a large number of components.

Start by constructing the components. Then construct a panel for each component group and set its layout manager if it is not a flow layout (the default for panels). Add a border to the panel if required. Finally, add the components to their panels. Continue in this fashion until you reach the outermost containers, which you add to the frame.

Here is an outline of the code required for the pizza user interface:

```

JPanel radioButtonPanel = new JPanel();
radioButtonPanel.setLayout(new GridLayout(3, 1));
radioButtonPanel.setBorder(new TitledBorder(new EtchedBorder(), "Size"));
radioButtonPanel.add(smallButton);
radioButtonPanel.add(mediumButton);
radioButtonPanel.add(largeButton);

JPanel checkBoxPanel = new JPanel();
checkBoxPanel.setLayout(new GridLayout(2, 1));
checkBoxPanel.add(pepperoniButton);
checkBoxPanel.add(anchoviesButton);

JPanel pricePanel = new JPanel(); // Uses FlowLayout by default
pricePanel.add(new JLabel("Your Price: "));
pricePanel.add(priceTextField);

```

```

JPanel centerPanel = new JPanel(); // Uses FlowLayout
centerPanel.add(radioButtonPanel);
centerPanel.add(checkBoxPanel);

// Frame uses BorderLayout by default
add(centerPanel, BorderLayout.CENTER);
add(pricePanel, BorderLayout.SOUTH);

```

Programming Tip 11.1



Use a GUI Builder

As you have seen, implementing even a simple graphical user interface in Java is quite tedious. You have to write a lot of code for constructing components, using layout managers, and providing event handlers. Most of the code is repetitive.

A GUI builder takes away much of the tedium. Most GUI builders help you in three ways:

- You drag and drop components onto a panel. The GUI builder writes the layout management code for you.
- You customize components with a dialog box, setting properties such as fonts, colors, text, and so on. The GUI builder writes the customization code for you.
- You provide event handlers by picking the event to process and providing just the code snippet for the listener method. The GUI builder writes the boilerplate code for attaching a listener object.

Java 6 introduced GroupLayout, a powerful layout manager that was specifically designed to be used by GUI builders. The free NetBeans development environment, available from <http://netbeans.org>, makes use of this layout manager—see Figure 7.

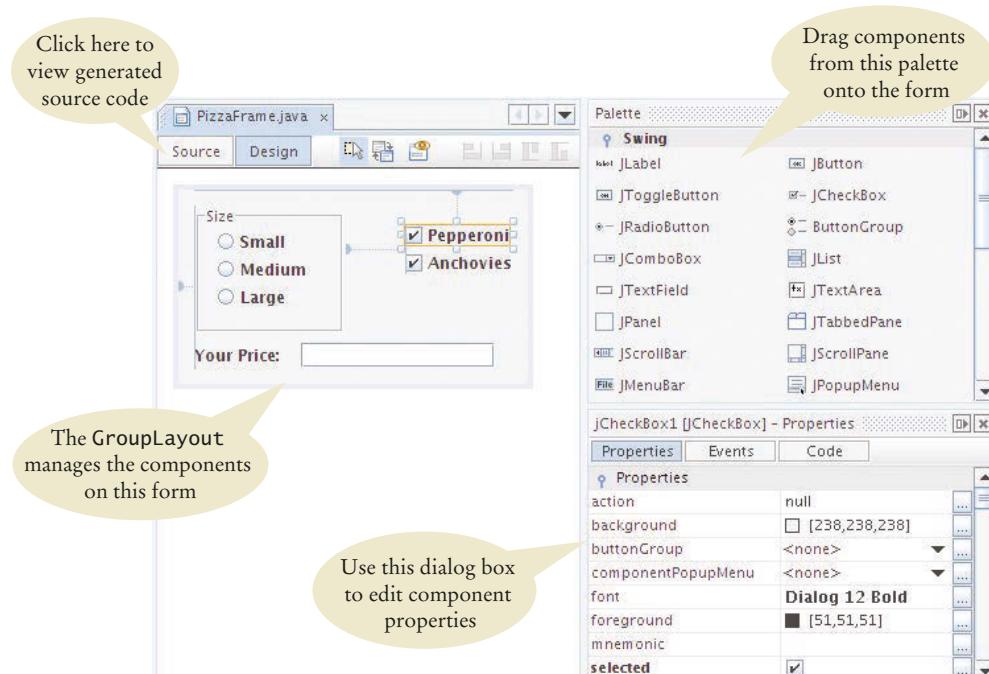


Figure 7 A GUI Builder

If you need to build a complex user interface, you will find that learning to use a GUI builder is a very worthwhile investment. You will spend less time writing boring code, and you will have more fun designing your user interface and focusing on the functionality of your program.



WORKED EXAMPLE 11.1

Programming a Working Calculator



In this Worked Example, we implement arithmetic and scientific operations for a calculator. The sample program in Section 11.1 showed how to lay out the buttons for a simple calculator, and we use that program as a starting point. Go to wiley.com/go/bjlo2examples and download Worked Example 11.1.

11.3 Menus

A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.

Anyone who has ever used a graphical user interface is familiar with pull-down menus (see Figure 8). At the top of the frame is a *menu bar* that contains the top-level menus. Each menu is a collection of *menu items* and *submenus*.

The sample program for this section builds up a small but typical menu and traps the action events from the menu items. The program allows the user to specify the font for a label by selecting a face name, font size, and font style. In Java it is easy to create these menus.

You add the menu bar to the frame:

```
public class MyFrame extends JFrame
{
    public MyFrame()
    {
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);
        . .
    }
    . .
}
```

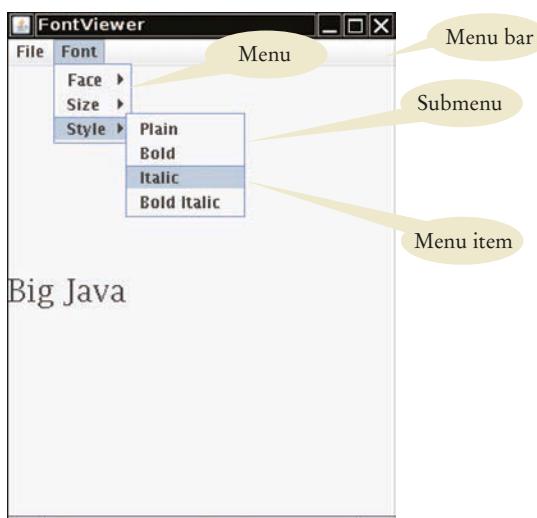


Figure 8
Pull-Down Menus

A menu provides a list of available choices.



© hillisphotography/Stockphoto

Menus are then added to the menu bar:

```
JMenu fileMenu = new JMenu("File");
JMenu fontMenu = new JMenu("Font");
menuBar.add(fileMenu);
menuBar.add(fontMenu);
```

You add menu items and submenus with the add method:

```
JMenuItem exitItem = new JMenuItem("Exit");
fileMenu.add(exitItem);

JMenu styleMenu = new JMenu("Style");
fontMenu.add(styleMenu); // A submenu
```

Menu items generate action events.

A menu item has no further submenus. When the user selects a menu item, the menu item sends an action event. Therefore, you want to add a listener to each menu item:

```
ActionListener listener = new ExitItemListener();
exitItem.addActionListener(listener);
```

You add action listeners only to menu items, not to menus or the menu bar. When the user clicks on a menu name and a submenu opens, no action event is sent.

To keep the program readable, it is a good idea to use a separate method for each menu or set of related menus. For example,

```
public JMenu createFaceMenu()
{
    JMenu menu = new JMenu("Face");
    menu.add(createFaceItem("Serif"));
    menu.add(createFaceItem("SansSerif"));
    menu.add(createFaceItem("Monospaced"));
    return menu;
}
```

Now consider the createFaceItem method. It has a string parameter variable for the name of the font face. When the item is selected, its action listener needs to

1. Set the current face name to the menu item text.
2. Make a new font from the current face, size, and style, and apply it to the label.

We have three menu items, one for each supported face name. Each of them needs to set a different name in the first step. Of course, we can make three listener classes `SerifListener`, `SansSerifListener`, and `MonospacedListener`, but that is not very elegant. After all, the actions only vary by a single string. We can store that string inside the listener class and then make three objects of the same listener class:

```
class FaceItemListener implements ActionListener
{
    private String name;

    public FaceItemListener(String newName) { name = newName; }
```

```

    public void actionPerformed(ActionEvent event)
    {
        facename = name; // Sets an instance variable of the frame class
        setLabelFont();
    }
}

```

Now we can install a listener object with the appropriate name:

```

public JMenuItem createFaceItem(String name)
{
    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new FaceItemListener(name);
    item.addActionListener(listener);
    return item;
}

```

This approach is still a bit tedious. We can do better by using a local inner class (see Special Topic 10.2). When we move the declaration of the inner class inside the `createFaceItem` method, the `actionPerformed` method can access the `name` parameter variable directly. Before Java 8, it was necessary to declare `name` as a `final` variable in order for it to be accessible from an inner class method.

```

public JMenuItem createFaceItem(final String name)
{
    class FaceItemListener implements ActionListener // A local inner class
    {
        public void actionPerformed(ActionEvent event)
        {
            facename = name; // Accesses the local variable name
            setLabelFont();
        }
    }

    JMenuItem item = new JMenuItem(name);
    ActionListener listener = new FaceItemListener();
    item.addActionListener(listener);
    return item;
}

```

The same strategy is used for the `createSizeItem` and `createStyleItem` methods.

sec03/FontViewer2.java

```

1 import javax.swing.JFrame;
2
3 /**
4  * This program uses a menu to display font effects.
5 */
6 public class FontViewer2
7 {
8     public static void main(String[] args)
9     {
10         JFrame frame = new FontFrame2();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setTitle("FontViewer");
13         frame.setVisible(true);
14     }
15 }

```

sec03/FontFrame2.java

```
 1 import java.awt.BorderLayout;
 2 import java.awt.Font;
 3 import java.awt.event.ActionEvent;
 4 import java.awt.event.ActionListener;
 5 import javax.swing.JFrame;
 6 import javax.swing.JLabel;
 7 import javax.swing.JMenu;
 8 import javax.swing.JMenuBar;
 9 import javax.swing.JMenuItem;
10
11 /**
12  * This frame has a menu with commands to change the font
13  * of a text sample.
14 */
15 public class FontFrame2 extends JFrame
16 {
17     private static final int FRAME_WIDTH = 300;
18     private static final int FRAME_HEIGHT = 400;
19
20     private JLabel label;
21     private String facename;
22     private int fontstyle;
23     private int fontsize;
24
25     /**
26      * Constructs the frame.
27     */
28     public FontFrame2()
29     {
30         // Construct text sample
31         label = new JLabel("Big Java");
32         add(label, BorderLayout.CENTER);
33
34         // Construct menu
35         JMenuBar menuBar = new JMenuBar();
36         setJMenuBar(menuBar);
37         menuBar.add(createFileMenu());
38         menuBar.add(createFontMenu());
39
40         facename = "Serif";
41         fontsize = 24;
42         fontstyle = Font.PLAIN;
43
44         setLabelFont();
45         setSize(FRAME_WIDTH, FRAME_HEIGHT);
46     }
47
48     class ExitItemListener implements ActionListener
49     {
50         public void actionPerformed(ActionEvent event)
51         {
52             System.exit(0);
53         }
54     }
55
56     /**
57      * Creates the File menu.
```

```
58     @return the menu
59 */
60 public JMenu createFileMenu()
61 {
62     JMenu menu = new JMenu("File");
63     JMenuItem exitItem = new JMenuItem("Exit");
64     ActionListener listener = new ExitItemListener();
65     exitItem.addActionListener(listener);
66     menu.add(exitItem);
67     return menu;
68 }
69
70 /**
71  * Creates the Font submenu.
72  * @return the menu
73 */
74 public JMenu createFontMenu()
75 {
76     JMenu menu = new JMenu("Font");
77     menu.add(createFaceMenu());
78     menu.add(createSizeMenu());
79     menu.add(createStyleMenu());
80     return menu;
81 }
82
83 /**
84  * Creates the Face submenu.
85  * @return the menu
86 */
87 public JMenu createFaceMenu()
88 {
89     JMenu menu = new JMenu("Face");
90     menu.add(createFaceItem("Serif"));
91     menu.add(createFaceItem("SansSerif"));
92     menu.add(createFaceItem("Monospaced"));
93     return menu;
94 }
95
96 /**
97  * Creates the Size submenu.
98  * @return the menu
99 */
100 public JMenu createSizeMenu()
101 {
102     JMenu menu = new JMenu("Size");
103     menu.add(createSizeItem("Smaller", -1));
104     menu.add(createSizeItem("Larger", 1));
105     return menu;
106 }
107
108 /**
109  * Creates the Style submenu.
110  * @return the menu
111 */
112 public JMenu createStyleMenu()
113 {
114     JMenu menu = new JMenu("Style");
115     menu.add(createStyleItem("Plain", Font.PLAIN));
116     menu.add(createStyleItem("Bold", Font.BOLD));
```

```
117     menu.add(createStyleItem("Italic", Font.ITALIC));
118     menu.add(createStyleItem("Bold Italic", Font.BOLD
119                         + Font.ITALIC));
120     return menu;
121 }
122
123 /**
124  * Creates a menu item to change the font face and set its action listener.
125  * @param name the name of the font face
126  * @return the menu item
127 */
128 public JMenuItem createFaceItem(final String name)
129 {
130     class FaceItemListener implements ActionListener
131     {
132         public void actionPerformed(ActionEvent event)
133         {
134             facename = name;
135             setLabelFont();
136         }
137     }
138
139     JMenuItem item = new JMenuItem(name);
140     ActionListener listener = new FaceItemListener();
141     item.addActionListener(listener);
142     return item;
143 }
144
145 /**
146  * Creates a menu item to change the font size
147  * and set its action listener.
148  * @param name the name of the menu item
149  * @param increment the amount by which to change the size
150  * @return the menu item
151 */
152 public JMenuItem createSizeItem(String name, final int increment)
153 {
154     class SizeItemListener implements ActionListener
155     {
156         public void actionPerformed(ActionEvent event)
157         {
158             fontsize = fontsize + increment;
159             setLabelFont();
160         }
161     }
162
163     JMenuItem item = new JMenuItem(name);
164     ActionListener listener = new SizeItemListener();
165     item.addActionListener(listener);
166     return item;
167 }
168
169 /**
170  * Creates a menu item to change the font style
171  * and set its action listener.
172  * @param name the name of the menu item
173  * @param style the new font style
174  * @return the menu item
175 */
```

```

176 public JMenuItem createStyleItem(String name, final int style)
177 {
178     class StyleItemListener implements ActionListener
179     {
180         public void actionPerformed(ActionEvent event)
181         {
182             fontstyle = style;
183             setLabelFont();
184         }
185     }
186
187     JMenuItem item = new JMenuItem(name);
188     ActionListener listener = new StyleItemListener();
189     item.addActionListener(listener);
190     return item;
191 }
192
193 /**
194 * Sets the font of the text sample.
195 */
196 public void setLabelFont()
197 {
198     Font f = new Font(facename, fontstyle, fontsize);
199     label.setFont(f);
200 }
201 }
```

SELF CHECK

12. Why do `JMenu` objects not generate action events?
13. Can you add a menu item directly to the menu bar? Try it out. What happens?
14. Why is the `increment` parameter variable in the `createSizeItem` method declared as `final`?
15. Why can't the `createFaceItem` method simply set the `facename` instance variable, like this:

```

class FaceItemListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        setLabelFont();
    }
}

public JMenuItem createFaceItem(String name)
{
    JMenuItem item = new JMenuItem(name);
    facename = name;
    ActionListener listener = new FaceItemListener();
    item.addActionListener(listener);
    return item;
}
```

16. In this program, the font specification (name, size, and style) is stored in instance variables. Why was this not necessary in the program of the previous section?

Practice It Now you can try these exercises at the end of the chapter: R11.12, E11.6, E11.7.

11.4 Exploring the Swing Documentation

You should learn to navigate the API documentation to find out more about user-interface components.

In the preceding sections, you saw the basic properties of the most common user-interface components. We purposefully omitted many options and variations to simplify the discussion. You can go a long way by using only the simplest properties of these components. If you want to implement a more sophisticated effect, you can look inside the Swing documentation. You may find the documentation intimidating at first glance, though. The purpose of this section is to show you how you can use the documentation to your advantage without being overwhelmed.

As an example, consider a program for mixing colors by specifying the red, green, and blue values. How can you specify the colors? Of course, you could supply three text fields, but sliders would be more convenient for users of your program (see Figure 9).

The Swing user-interface toolkit has a large set of user-interface components. How do you know if there is a slider? You can buy a book that illustrates all Swing components. Or you can run the sample application included in the Java Development Kit that shows off all Swing components (see Figure 10). Or you can look at the names of all of the classes that start with `J` and decide that `JSlider` may be a good candidate.

Next, you need to ask yourself a few questions:

- How do I construct a `JSlider`?
- How can I get notified when the user has moved it?
- How can I tell to which value the user has set it?

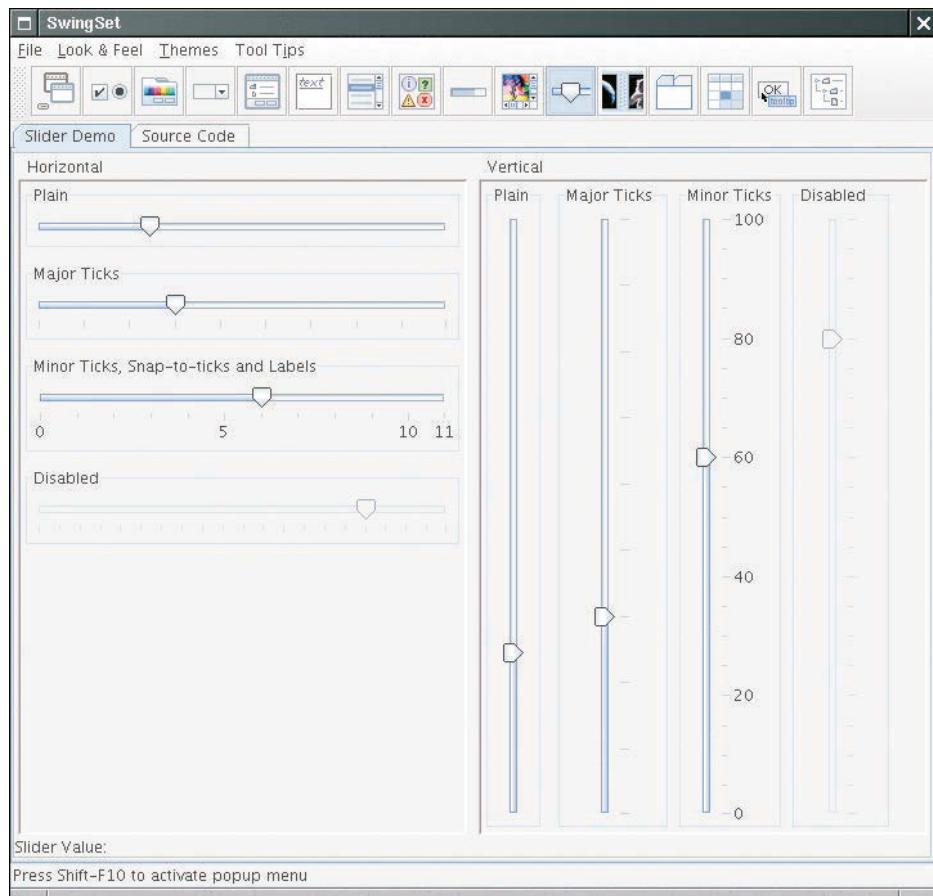


Figure 9 A Color Viewer with Sliders



© René Mansi/Stockphoto.

Figure 10
The SwingSet Demo



When you look at the documentation of the `JSlider` class, you will probably not be happy. There are over 50 methods in the `JSlider` class and over 250 inherited methods, and some of the method descriptions look downright scary, such as the one in Figure 11. Apparently some folks out there are concerned about the `valueIsAdjusting` property, whatever that may be, and the designers of this class felt it necessary to supply a method to tweak that property. Until you too feel that need, your best bet is

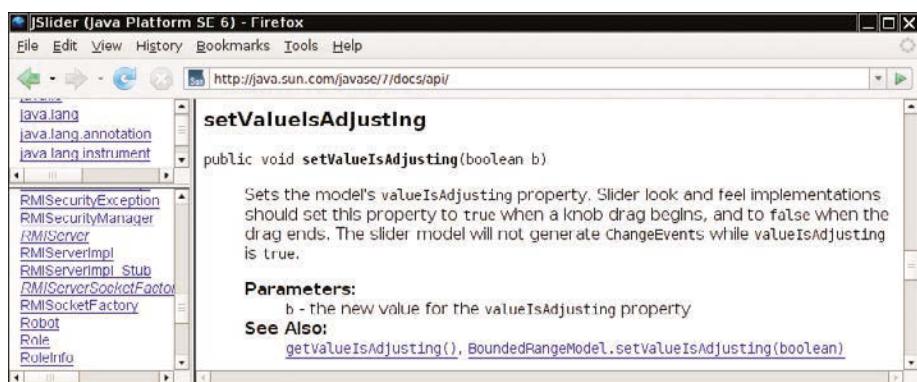


Figure 11 A Mysterious Method Description from the API Documentation

to ignore this method. As the author of an introductory book, it pains me to tell you to ignore certain facts. But the truth of the matter is that the Java library is so large and complex that nobody understands it in its entirety, not even the designers of Java themselves. You need to develop the ability to separate fundamental concepts from ephemeral minutiae. For example, it is important that you understand the concept of event handling. Once you understand the concept, you can ask the question, “What event does the slider send when the user moves it?” But it is not important that you memorize how to set tick marks or that you know how to implement a slider with a custom look and feel.

Let’s go back to our fundamental questions. In Java 6, there are six constructors for the `JSlider` class. You want to learn about one or two of them. You must strike a balance somewhere between the trivial and the bizarre. Consider

`public JSlider()`

Creates a horizontal slider with the range 0 to 100 and an initial value of 50.

Maybe that is good enough for now, but what if you want another range or initial value? It seems too limited.

On the other side of the spectrum, there is

`public JSlider(BoundedRangeModel brm)`

Creates a horizontal slider using the specified `BoundedRangeModel`.

Whoa! What is that? You can click on the `BoundedRangeModel` link to get a long explanation of this class. This appears to be some internal mechanism for the Swing implementors. Let’s try to avoid this constructor if we can. Looking further, we find

`public JSlider(int min, int max, int value)`

Creates a horizontal slider using the specified `min`, `max`, and `value`.

This sounds general enough to be useful and simple enough to be usable. You might want to stash away the fact that you can have vertical sliders as well.

Next, you want to know what events a slider generates. There is no `addActionListener` method. That makes sense. Adjusting a slider seems different from clicking a button, and Swing uses a different event type for these events. There is a method

`public void addChangeListener(ChangeListener l)`

Click on the `ChangeListener` link to find out more about this interface. It has a single method

`void stateChanged(ChangeEvent e)`

Apparently, that method is called whenever the user moves the slider. What is a `ChangeEvent`? Once again, click on the link, to find out that this event class has *no* methods of its own, but it inherits the `getSource` method from its superclass `EventObject`. The `getSource` method tells us which component generated this event, but we don’t need that information—we know that the event came from the slider.

Now let’s make a plan: Add a change event listener to each slider. When the slider is changed, the `stateChanged` method is called. Find out the new value of the slider. Recompute the color value and repaint the color panel. That way, the color panel is continually repainted as the user moves one of the sliders.

To compute the color value, you will still need to get the current value of the slider. Look at all the methods that start with `get`. Sure enough, you find

`public int getValue()`

Returns the slider’s value.

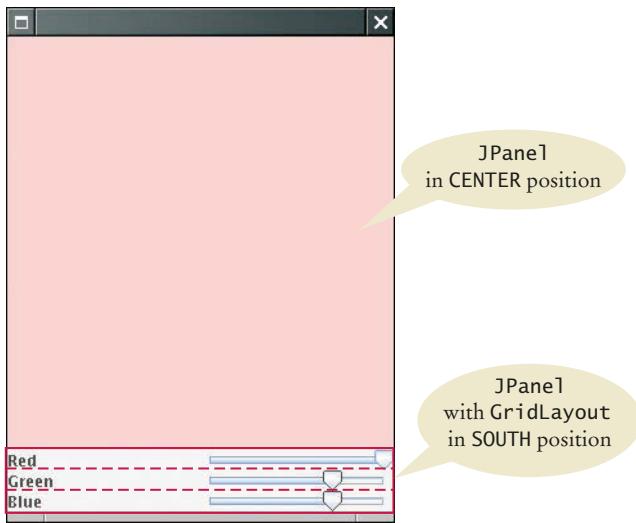


Figure 12 The Components of the Color Viewer Frame

Now you know everything you need to write the program. The program uses one new Swing component and one event listener of a new type. After having mastered the basics, you may want to explore the capabilities of the component further, for example by adding tick marks—see Exercise E11.9.

Figure 12 shows how the components are arranged in the frame.

sec04/ColorViewer.java

```

1 import javax.swing.JFrame;
2
3 public class ColorViewer
4 {
5     public static void main(String[] args)
6     {
7         JFrame frame = new ColorFrame();
8         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9         frame.setVisible(true);
10    }
11 }
```

sec04/ColorFrame.java

```

1 import java.awt.BorderLayout;
2 import java.awt.Color;
3 import java.awt.GridLayout;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;
6 import javax.swing.JPanel;
7 import javax.swing.JSlider;
8 import javax.swing.event.ChangeListener;
9 import javax.swing.event.ChangeEvent;
10
11 public class ColorFrame extends JFrame
12 {
13     private static final int FRAME_WIDTH = 300;
14     private static final int FRAME_HEIGHT = 400;
```

```
15     private JPanel colorPanel;
16     private JSlider redSlider;
17     private JSlider greenSlider;
18     private JSlider blueSlider;
19
20
21     public ColorFrame()
22     {
23         colorPanel = new JPanel();
24
25         add(colorPanel, BorderLayout.CENTER);
26         createControlPanel();
27         setSampleColor();
28         setSize(FRAME_WIDTH, FRAME_HEIGHT);
29     }
30
31     class ColorListener implements ChangeListener
32     {
33         public void stateChanged(ChangeEvent event)
34         {
35             setSampleColor();
36         }
37     }
38
39     public void createControlPanel()
40     {
41         ChangeListener listener = new ColorListener();
42
43         redSlider = new JSlider(0, 255, 255);
44         redSlider.addChangeListener(listener);
45
46         greenSlider = new JSlider(0, 255, 175);
47         greenSlider.addChangeListener(listener);
48
49         blueSlider = new JSlider(0, 255, 175);
50         blueSlider.addChangeListener(listener);
51
52         JPanel controlPanel = new JPanel();
53         controlPanel.setLayout(new GridLayout(3, 2));
54
55         controlPanel.add(new JLabel("Red"));
56         controlPanel.add(redSlider);
57
58         controlPanel.add(new JLabel("Green"));
59         controlPanel.add(greenSlider);
60
61         controlPanel.add(new JLabel("Blue"));
62         controlPanel.add(blueSlider);
63
64         add(controlPanel, BorderLayout.SOUTH);
65     }
66
67     /**
68      * Reads the slider values and sets the panel to
69      * the selected color.
70     */
71     public void setSampleColor()
72     {
73         // Read slider values
74     }
```

```

75     int red = redSlider.getValue();
76     int green = greenSlider.getValue();
77     int blue = blueSlider.getValue();
78
79     // Set panel background to selected color
80
81     colorPanel.setBackground(new Color(red, green, blue));
82     colorPanel.repaint();
83 }
84
}

```

SELF CHECK

- 17.** Suppose you want to allow users to pick a color from a color dialog box. Which class would you use? Look in the API documentation.
- 18.** Why does a slider emit change events and not action events?

Practice It

Now you can try these exercises at the end of the chapter: R11.14, E11.2, E11.9.

11.5 Using Timer Events for Animations

In this section we introduce timer events and show how you can use them to implement simple animations.

The `Timer` class in the `javax.swing` package generates a sequence of action events, spaced at even time intervals. (You can think of a timer as an invisible button that is automatically clicked.) This is useful whenever you want to send continuous updates to a component. For example, in an animation, you may want to update a scene ten times per second and redisplay the image to give the illusion of movement.

When you use a timer, you specify the frequency of the events and an object of a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Finally, start the timer.

```

class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        Action that is executed at each timer event.
    }
}

MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();

```

Then the timer calls the `actionPerformed` method of the `listener` object every `interval` milliseconds.

A timer generates action events at fixed intervals.



A Swing timer notifies a listener with each “tick”.

Our sample program will display a moving rectangle. We first supply a RectangleComponent class with a moveRectangleBy method that moves the rectangle by a given amount.

sec05/RectangleComponent.java

```

1 import java.awt.Graphics;
2 import javax.swing.JComponent;
3
4 /**
5  * This component displays a rectangle that can be moved.
6 */
7 public class RectangleComponent extends JComponent
8 {
9     private static final int RECTANGLE_WIDTH = 20;
10    private static final int RECTANGLE_HEIGHT = 30;
11
12    private int xLeft;
13    private int yTop;
14
15    public RectangleComponent()
16    {
17        xLeft = 0;
18        yTop = 0;
19    }
20
21    public void paintComponent(Graphics g)
22    {
23        g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
24    }
25
26    /**
27     * Moves the rectangle by a given amount.
28     * @param dx the amount to move in the x-direction
29     * @param dy the amount to move in the y-direction
30     */
31    public void moveRectangleBy(int dx, int dy)
32    {
33        xLeft = xLeft + dx;
34        yTop = yTop + dy;
35        repaint();
36    }
37}

```

To make an animation, the timer listener should update and repaint a component several times per second.

Note the call to repaint in the moveRectangleBy method. This call is necessary to ensure that the component is repainted after the position of the rectangle has been changed. The call to repaint forces a call to the paintComponent method. The paintComponent method redraws the component, causing the rectangle to appear at the updated location.

The actionPerformed method of the timer listener moves the rectangle one pixel down and to the right:

```
scene.moveRectangleBy(1, 1);
```

Because the actionPerformed method is called many times per second, the rectangle appears to move smoothly across the frame.

sec05/RectangleFrame.java

```

1 import java.awt.event.ActionEvent;
2 import java.awt.event.ActionListener;
3 import javax.swing.JFrame;
4 import javax.swing.Timer;
5
6 /**
7  * This frame contains a moving rectangle.
8 */
9 public class RectangleFrame extends JFrame
10 {
11     private static final int FRAME_WIDTH = 300;
12     private static final int FRAME_HEIGHT = 400;
13
14     private RectangleComponent scene;
15
16     class TimerListener implements ActionListener
17     {
18         public void actionPerformed(ActionEvent event)
19         {
20             scene.moveRectangleBy(1, 1);
21         }
22     }
23
24     public RectangleFrame()
25     {
26         scene = new RectangleComponent();
27         add(scene);
28         setSize(FRAME_WIDTH, FRAME_HEIGHT);
29         ActionListener listener = new TimerListener();
30         final int DELAY = 100; // Milliseconds between timer ticks
31         Timer t = new Timer(DELAY, listener);
32         t.start();
33     }
34 }
```



- 19.** Why does a timer require a listener object?
- 20.** How can you make the rectangle move backwards?
- 21.** Describe two ways of modifying the program so that the rectangle moves twice as fast.
- 22.** How can you make a car move instead of a rectangle?
- 23.** How can you make two rectangles move in parallel in the scene?
- 24.** What would happen if you omitted the call to repaint in the moveRectangleBy method?

Practice It Now you can try these exercises at the end of the chapter: E11.10, E11.11, E11.12.

11.6 Mouse Events

You use a mouse listener to capture mouse events.

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to listen to mouse events. Mouse listeners are more complex than action listeners, the listeners that process button clicks and timer ticks.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

```
public interface MouseListener
{
    void mousePressed(MouseEvent event);
        // Called when a mouse button has been pressed on a component
    void mouseReleased(MouseEvent event);
        // Called when a mouse button has been released on a component
    void mouseClicked(MouseEvent event);
        // Called when the mouse has been clicked on a component
    void mouseEntered(MouseEvent event);
        // Called when the mouse enters a component
    void mouseExited(MouseEvent event);
        // Called when the mouse exits a component
}
```

The `mousePressed` and `mouseReleased` methods are called whenever a mouse button is pressed or released. If a button is pressed and released in quick succession, and the mouse has not moved, then the `mouseClicked` method is called as well. The `mouseEntered` and `mouseExited` methods can be used to highlight a user-interface component whenever the mouse is pointing inside it.

The most commonly used method is `mousePressed`. Users generally expect that their actions are processed as soon as the mouse button is pressed.

You add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        Process mouse event at (x, y).
    }

    // Do-nothing methods
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
}

MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

In our sample program, a user clicks on a component containing a rectangle. Whenever the mouse button is pressed, the rectangle is moved to the mouse location. We



© James Brey/iStockphoto.

In Swing, a mouse event isn't a gathering of rodents; it's notification of a mouse click by the program user.

first enhance the RectangleComponent class and add a `moveRectangleTo` method to move the rectangle to a new position.

sec06/RectangleComponent2.java

```

1 import java.awt.Graphics;
2 import java.awt.Rectangle;
3 import javax.swing.JComponent;
4
5 /**
6  * This component displays a rectangle that can be moved.
7 */
8 public class RectangleComponent2 extends JComponent
9 {
10     private static final int RECTANGLE_WIDTH = 20;
11     private static final int RECTANGLE_HEIGHT = 30;
12
13     private int xLeft;
14     private int yTop;
15
16     public RectangleComponent2()
17     {
18         xLeft = 0;
19         yTop = 0;
20     }
21
22     public void paintComponent(Graphics g)
23     {
24         g.fillRect(xLeft, yTop, RECTANGLE_WIDTH, RECTANGLE_HEIGHT);
25     }
26
27     /**
28      * Moves the rectangle to the given location.
29      * @param x the x-position of the new location
30      * @param y the y-position of the new location
31     */
32     public void moveRectangleTo(int x, int y)
33     {
34         xLeft = x;
35         yTop = y;
36         repaint();
37     }
38 }
```

Note the call to `repaint` in the `moveRectangleTo` method. As you saw before, this call causes the component to repaint itself and show the rectangle in the new position.

Now, add a mouse listener to the component. Whenever the mouse is pressed, the listener moves the rectangle to the mouse location.

```

class MousePressListener implements MouseListener
{
    public void mousePressed(MouseEvent event)
    {
        int x = event.getX();
        int y = event.getY();
        scene.moveRectangleTo(x, y);
    }
    . .
}
```

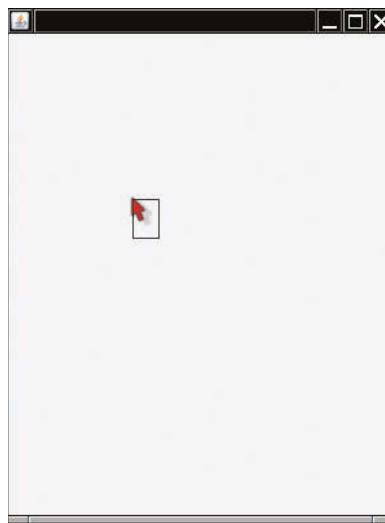


Figure 13 Clicking the Mouse Moves the Rectangle

It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all five methods of the interface must be implemented. The unused methods are simply implemented as do-nothing methods.

Go ahead and run the `RectangleViewer2` program. Whenever you click the mouse inside the frame, the top-left corner of the rectangle moves to the mouse pointer (see Figure 13).

sec06/RectangleViewer2.java

```

1 import javax.swing.JFrame;
2
3 /**
4  * This program displays a rectangle that can be moved with the mouse.
5 */
6 public class RectangleViewer2
7 {
8     public static void main(String[] args)
9     {
10         JFrame frame = new RectangleFrame2();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setVisible(true);
13     }
14 }
```

sec06/RectangleFrame2.java

```

1 import java.awt.event.MouseListener;
2 import java.awt.event.MouseEvent;
3 import javax.swing.JFrame;
4
5 /**
6  * This frame contains a moving rectangle.
7 */
8 public class RectangleFrame2 extends JFrame
9 {
```

```

10     private static final int FRAME_WIDTH = 300;
11     private static final int FRAME_HEIGHT = 400;
12
13     private RectangleComponent2 scene;
14
15     class MousePressListener implements MouseListener
16     {
17         public void mousePressed(MouseEvent event)
18         {
19             int x = event.getX();
20             int y = event.getY();
21             scene.moveRectangleTo(x, y);
22         }
23
24         // Do-nothing methods
25         public void mouseReleased(MouseEvent event) {}
26         public void mouseClicked(MouseEvent event) {}
27         public void mouseEntered(MouseEvent event) {}
28         public void mouseExited(MouseEvent event) {}
29     }
30
31     public RectangleFrame2()
32     {
33         scene = new RectangleComponent2();
34         add(scene);
35         MouseListener listener = new MousePressListener();
36         scene.addMouseListener(listener);
37
38         setSize(FRAME_WIDTH, FRAME_HEIGHT);
39     }
40 }
```

SELF CHECK

25. Why was the `moveRectangleBy` method in `RectangleComponent2` replaced with a `moveRectangleTo` method?
26. Why must the `MousePressListener` class supply five methods?
27. How could you change the behavior of the program so that a new rectangle is added whenever the mouse is clicked?

Practice It

Now you can try these exercises at the end of the chapter: R11.21, P11.10, P11.11.

Special Topic 11.1**Keyboard Events**

If you program a game, you may want to process keystrokes, such as the arrow keys. Add a key listener to the component on which you draw the game scene. The `KeyListener` interface has three methods. As with a mouse listener, you are most interested in key press events, and you can leave the other two methods empty. Your key listener class should look like this:

```

class MyKeyListener implements KeyListener
{
    public void keyPressed(KeyEvent event)
    {
        String key = KeyStroke.getKeyStrokeForEvent(event).toString();
        key = key.replace("pressed ", "");
        Process key.
```

```

    }

    // Do-nothing methods
    public void keyReleased(KeyEvent event) {}
    public void keyTyped(KeyEvent event) {}
}

```

The call `KeyStroke.getKeyStrokeForEvent(event).toString()` turns the event object into a text description of the key, such as "pressed LEFT". In the next line, we eliminate the "pressed " prefix. The remainder is a string such as "LEFT" or "A" that describes the key that was pressed. You can find a list of all key names in the API documentation of the `KeyStroke` class.

As always, remember to attach the listener to the event source:

```
KeyListener listener = new MyKeyListener();
scene.addKeyListener(listener);
```

In order to receive key events, your component must call

```
scene.setFocusable(true);
```



© Dmitry Shironosov/
iStockphoto.

FULL CODE EXAMPLE

Go to wiley.com/go/bj1o2code to download a complete program that uses the arrow keys to move a rectangle.

Special Topic 11.2



Event Adapters

In the preceding section you saw how to install a mouse listener in a mouse event source and how the listener methods are called when an event occurs. Usually, a program is not interested in all listener notifications. For example, a program may only be interested in mouse clicks and may not care that these mouse clicks are composed of "mouse pressed" and "mouse released" events. Of course, the program could supply a listener that declares all those methods in which it has no interest as "do-nothing" methods, for example:

```
class MouseClickListener implements MouseListener
{
    public void mouseClicked(MouseEvent event)
    {
        Mouse click action
    }

    // Four do-nothing methods
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mousePressed(MouseEvent event) {}
    public void mouseReleased(MouseEvent event)
}
```

To avoid this labor, some friendly soul has created a `MouseListener` class that implements the `MouseListener` interface such that all methods do nothing. You can *extend* that class, inheriting the do-nothing methods and overriding the methods that you care about, like this:

```
class MouseClickListener extends MouseAdapter
{
    public void mouseClicked(MouseEvent event)
    {
        Mouse click action
    }
}
```

There is also a `KeyAdapter` that implements the `KeyListener` interface (see Special Topic 11.1), providing three do-nothing methods.

**WORKED EXAMPLE 11.2****Adding Mouse and Keyboard Support to the Bar Chart Creator**

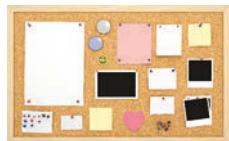
In this Worked Example, we will enhance the bar chart creator of Worked Example 10.1 and add support for mouse and keyboard operations. Go to wiley.com/go/bj102examples and download Worked Example 11.2.

**VIDEO EXAMPLE 11.1****Designing a Baby Naming Program**

In this Video Example, you will see how to design a user interface for a program that suggests baby names. Go to wiley.com/go/bj102videos to view Video Example 11.1.



© Nancy Ross/
iStockphoto.

CHAPTER SUMMARY**Learn how to arrange multiple components in a container.**

- User-interface components are arranged by placing them inside containers. Containers can be placed inside larger containers.
- Each container has a layout manager that directs the arrangement of its components.
- Three useful layout managers are the border layout, flow layout, and grid layout.
- When adding a component to a container with the border layout, specify the NORTH, SOUTH, WEST, EAST, or CENTER position.
- The content pane of a frame has a border layout by default. A panel has a flow layout by default.

Select among the Swing components for presenting choices to the user.

- For a small set of mutually exclusive choices, use a group of radio buttons or a combo box.
- Add radio buttons to a `ButtonGroup` so that only one button in the group is selected at any time.
- You can place a border around a panel to group its contents visually.
- For a binary choice, use a check box.
- For a large set of choices, use a combo box.
- Radio buttons, check boxes, and combo boxes generate action events, just as buttons do.

Implement menus in a Swing program.

- A frame contains a menu bar. The menu bar contains menus. A menu contains submenus and menu items.
- Menu items generate action events.



Use the Swing documentation.

- You should learn to navigate the API documentation to find out more about user-interface components.

**Use timer events to implement animations.**

- A timer generates action events at fixed intervals.
- To make an animation, the timer listener should update and repaint a component several times per second.

**Write programs that process mouse events.**

- You use a mouse listener to capture mouse events.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

<code>java.awt.BorderLayout</code>	<code>javax.swing.ButtonGroup</code>
<code>CENTER</code>	<code>add</code>
<code>EAST</code>	<code>javax.swing.JCheckBox</code>
<code>NORTH</code>	<code>javax.swing.JComboBox</code>
<code>SOUTH</code>	<code>addItem</code>
<code>WEST</code>	<code>getSelectedItem</code>
<code>java.awt.Component</code>	<code>isEditable</code>
<code>addKeyListener</code>	<code>setEditable</code>
<code>addMouseListener</code>	<code> setSelectedItem</code>
<code>setFocusable</code>	<code>javax.swing.JComponent</code>
<code>java.awt.Container</code>	<code>setBorder</code>
<code>setLayout</code>	<code>setFocusable</code>
<code>java.awt.FlowLayout</code>	<code>setFont</code>
<code>java.awt.Font</code>	<code>javax.swing.JFrame</code>
<code>BOLD</code>	<code>setJMenuBar</code>
<code>ITALIC</code>	<code>javax.swing.JMenu</code>
<code>java.awt.GridLayout</code>	<code>add</code>
<code>java.awt.event.KeyEvent</code>	<code>javax.swing.JMenuBar</code>
<code>java.awt.event.KeyListener</code>	<code>add</code>
<code>keyPressed</code>	<code>javax.swing.JMenuItem</code>
<code>keyReleased</code>	<code>javax.swing.JRadioButton</code>
<code>keyTyped</code>	<code>javax.swing.JSlider</code>
<code>java.awt.event.MouseEvent</code>	<code>addChangeListener</code>
<code>getX</code>	<code>getValue</code>
<code>getY</code>	<code>javax.swing.KeyStroke</code>
<code>java.awt.event.MouseListener</code>	<code>getKeyStrokeForEvent</code>
<code>mouseClicked</code>	<code>javax.swing.Timer</code>
<code>mouseEntered</code>	<code>start</code>
<code>mouseExited</code>	<code>stop</code>
<code>mousePressed</code>	<code>javax.swing.border.EtchedBorder</code>
<code>mouseReleased</code>	<code>javax.swing.border.TitledBorder</code>
<code>javax.swing.AbstractButton</code>	<code>javax.swing.event.ChangeEvent</code>
<code>isSelected</code>	<code>javax.swing.event.ChangeListener</code>
<code>setSelected</code>	<code>stateChanged</code>

REVIEW EXERCISES

- **R11.1** Can you use a flow layout for the components in a frame? If yes, how?
- **R11.2** What is the advantage of a layout manager over telling the container “place this component at position (x, y) ”?
- **R11.3** What happens when you place a single button into the CENTER area of a container that uses a border layout? Try it out by writing a small sample program if you aren’t sure of the answer.
- **R11.4** What happens if you place multiple buttons directly into the SOUTH area, without using a panel? Try it out by writing a small sample program if you aren’t sure of the answer.
- **R11.5** What happens when you add a button to a container that uses a border layout and omit the position? Try it out and explain.
- **R11.6** What happens when you try to add a button to another button? Try it out and explain.
- **R11.7** The control panel in Section 11.4 uses a grid layout manager. Explain a drawback of the grid that is apparent in Figure 12. What could you do to overcome this drawback?
- **R11.8** What is the difference between the grid layout and the grid bag layout?
- **R11.9** Can you add icons to check boxes, radio buttons, and combo boxes? Browse the Java documentation to find out. Then write a small test program to verify your findings.
- **R11.10** What is the difference between radio buttons and check boxes?
- **R11.11** Why do you need a button group for radio buttons but not for check boxes?
- **R11.12** What is the difference between a menu bar, a menu, and a menu item?
- **R11.13** When browsing through the Java documentation for more information about sliders, we ignored the `JSlider` constructor with no arguments. Why? Would it have worked in our sample program?
- **R11.14** How do you construct a vertical slider? Consult the Swing documentation for an answer.
- **R11.15** Why doesn’t a `JComboBox` send out change events?
- **R11.16** What component would you use to show a set of choices, as in a combo box, but so that several items are visible at the same time? Run the Swing demo application or look at a book with Swing example programs to find the answer.
- **R11.17** How many Swing user-interface components are there? Look at the Java documentation to get an approximate answer.
- **R11.18** How many methods does the `JProgressBar` component have? Be sure to count inherited methods. Look at the Java documentation.
- **R11.19** What is the difference between an `ActionEvent` and a `MouseEvent`?
- **R11.20** What information does an action event object carry? What additional information does a mouse event object carry? Hint: Check the API documentation.

- R11.21 Why does the `ActionListener` interface have only one method, whereas the `MouseListener` has five methods?

PRACTICE EXERCISES

- E11.1 Write an application with three buttons labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.
- E11.2 Add icons to the buttons of Exercise E11.1. Use a `JButton` constructor with an `Icon` argument and supply an `ImageIcon`.
- E11.3 Write an application with three radio buttons labeled “Red”, “Green”, and “Blue” that changes the background color of a panel in the center of the frame to red, green, or blue.
- E11.4 Write an application with three check boxes labeled “Red”, “Green”, and “Blue” that adds a red, green, or blue component to the background color of a panel in the center of the frame. This application can display a total of eight color combinations.
- E11.5 Write an application with a combo box containing three items labeled “Red”, “Green”, and “Blue” that change the background color of a panel in the center of the frame to red, green, or blue.
- E11.6 Write an application with a Color menu and menu items labeled “Red”, “Green”, and “Blue” that change the background color of a panel in the center of the frame to red, green, or blue.
- E11.7 Write a program that displays a number of rectangles at random positions. Supply menu items “Fewer” and “More” that generate fewer or more random rectangles. Each time the user selects “Fewer”, the count should be halved. Each time the user clicks on “More”, the count should be doubled.
- E11.8 Modify the program of Exercise E11.7 to replace the buttons with a slider to generate more or fewer random rectangles.
- E11.9 Modify the slider program in Section 11.4 to add a set of tick marks to each slider that show the exact slider position.
- E11.10 Write a program that uses a timer to print the current time once a second. *Hint:* The following code prints the current time:

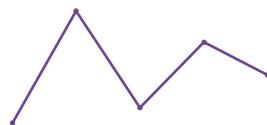

```
Date now = new Date();
System.out.println(now);
```

The `Date` class is in the `java.util` package.
- E11.11 Change the `RectangleComponent` for the animation in Section 11.5 so that the rectangle bounces off the edges of the component rather than simply moving outside.
- E11.12 Change the rectangle animation in Section 11.5 so that it shows two rectangles moving in opposite directions.

PROGRAMMING PROJECTS

... P11.1 Enhance the font viewer program to allow the user to select different font faces. Research the API documentation to find out how to find the available fonts on the user's system.

... P11.2 Write a program that lets users design charts such as the following:



Use appropriate components to ask for the x - and y -positions of the points, and to redraw the chart when the user adds an item.

P11.3 Write a program that animates a car so that it moves across a frame.

... P11.4 Write a program that animates two cars moving across a frame in opposite directions (but at different heights so that they don't collide.)

... P11.5 Write a program that displays a scrolling message in a panel. Use a timer for the scrolling effect. In the timer's action listener, move the starting position of the message and repaint. When the message has left the window, reset the starting position to the other corner. Provide a user interface to customize the message text, font, foreground and background colors, and the scrolling speed and direction.

P11.6 Change the RectangleComponent for the mouse listener program in Section 11.6 so that a new rectangle is added to the component whenever the mouse is clicked. *Hint:* Store all points on which the user clicked, and draw all rectangles in the paintComponent method.

P11.7 Write a program that prompts the user to enter the x - and y -positions of a center point and a radius, using text fields. When the user clicks a "Draw" button, draw a circle with that center and radius in a component.

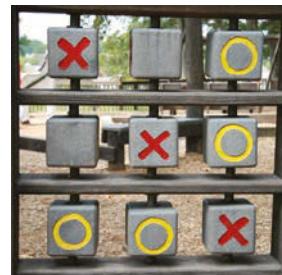
P11.8 Write a program that allows the user to specify a circle by typing the radius in a text field and then clicking on the center. Note that you don't need a "Draw" button.

P11.9 Write a program that allows the user to specify a circle with two mouse presses, the first one on the center and the second on a point on the periphery.

Hint: In the mouse press handler, you must keep track of whether you already received the center point in a previous mouse press.

P11.10 Write a program that allows the user to specify a triangle with three mouse presses. After the first mouse press, draw a small dot. After the second mouse press, draw a line joining the first two points. After the third mouse press, draw the entire triangle. The fourth mouse press erases the old triangle and starts a new one.

- P11.11** Implement a program that allows two players to play tic-tac-toe. Draw the game grid and an indication of whose turn it is (X or O). Upon the next click, check that the mouse click falls into an empty location, fill the location with the mark of the current player, and give the other player a turn. If the game is won, indicate the winner. Also supply a button for starting over.
- P11.12** Write a program that lets users design line charts with a mouse. When the user clicks above or below an existing point, the point is moved. (Allow for a few pixels of tolerance.) When the user clicks elsewhere, a point is added to the chart. When the user clicks on an existing point, and then a “Delete” button, the point is removed.
- Business P11.13** Write a program with a graphical interface that allows the user to convert an amount of money between U.S. dollars (USD), euros (EUR), and British pounds (GBP). The user interface should have the following elements: a text box to enter the amount to be converted, two combo boxes to allow the user to select the currencies, a button to make the conversion, and a label to show the result. Display a warning if the user does not choose different currencies. You can get up-to-date exchange rates from <http://www.ecb.europa.eu/stats/exchange/eurofxref/html/index.en.html>, or download from <http://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml> in a format that is easier to parse.
- Business P11.14** Write a program with a graphical interface that implements a login window with text fields for the user name and password. When the login is successful, hide the login window and open a new window with a welcome message. Follow these rules for validating the password:
1. The user name is not case sensitive.
 2. The password is case sensitive.
 3. The user has three opportunities to enter valid credentials.
- Otherwise, display an error message and terminate the program. When the program starts, read the file `users.txt`. Each line in that file contains a username and password, separated by a space. You should make a `users.txt` file for testing your program.
- Business P11.15** In Exercise P11.14, the password is shown as it is typed. Browse the Swing documentation to find an appropriate component for entering a password. Improve the solution of Exercise P11.14 by using this component instead of a text field. Each time the user types a letter, show a ■ character.



© Kathy Muller/iStockphoto.

ANSWERS TO SELF-CHECK QUESTIONS

1. Only the second one is displayed.
2. First add them to a panel, then add the panel to the north end of a frame.
3. Place them inside a panel with a `GridLayout` that has three rows and one column.
4. The button in the north stretches horizontally to fill the width of the frame. The height of the northern area is the normal height. The center button fills the remainder of the window.
5. To get the double-wide button, put it in the south of a panel with border layout whose center has a 3×2 grid layout with the keys 7, 8, 4, 5, 1, 2. Put that panel in the west of another border layout panel whose eastern area has a 4×1 grid layout with the remaining keys.
6. If you have many options, a set of radio buttons takes up a large area. A combo box can show many options without using up much space. But the user cannot see the options as easily.
7. If one of them is checked, the other one is unchecked. You should use radio buttons if that is the behavior you want.
8. You can't nest borders, but you can nest panels with borders:


```
JPanel p1 = new JPanel();
p1.setBorder(new EtchedBorder());
JPanel p2 = new JPanel();
p2.setBorder(new EtchedBorder());
p1.add(p2);
```
9. When any of the component settings is changed, the program simply queries all of them and updates the label.
10. To keep it from growing too large. It would have grown to the same width and height as the two panels below it.
11. Instead of using radio buttons with two choices, use a checkbox.
12. When you open a menu, you have not yet made a selection. Only `JMenuItem` objects correspond to selections.
13. Yes, you can—`JMenuItem` is a subclass of `JMenu`. The item shows up on the menu bar. When you click on it, its listener is called. But the

behavior feels unnatural for a menu bar and is likely to confuse users.

14. Using the `final` modifier enables the code to compile with versions prior to Java 8. In those versions, it was necessary to declare the parameter variable as `final` so that it could be accessed in a method of an inner class.
15. Then the `facename` variable is set when the menu item is added to the menu, not when the user selects the menu.
16. In the previous program, the user-interface components effectively served as storage for the font specification. Their current settings were used to construct the font. But a menu doesn't save settings; it just generates an action.
17. `JColorChooser`.
18. Action events describe one-time changes, such as button clicks. Change events describe continuous changes.
19. The timer needs to call some method whenever the time interval expires. It calls the `actionPerformed` method of the `listener` object.
20. Call `scene.moveRectangleBy(-1, -1)` in the `actionPerformed` method.
21. You can cut the timer delay in half (to 50 milliseconds between ticks), or you can double the distance by which the rectangle moves, by calling `scene.moveRectangleBy(2, 2)`.
22. The component class would need to draw a car at position (x, y) instead of a rectangle.
23. There are two entirely different ways:
 - a. Add a second `RectangleComponent` to the frame, using a grid layout. Change the `actionPerformed` method of the `TimerListener` to call `moveRectangleBy` on both components.
 - b. Draw a second rectangle in the `paintComponent` method of `RectangleComponent`.
24. The moved rectangles won't be painted, and the rectangle will appear to be stationary until the frame is repainted for an external reason.
25. Because you know the current mouse position, not the amount by which the mouse has moved.

26. It implements the `MouseListener` interface, which has five methods.
27. The `RectangleComponent2` class needs to keep track of the locations of multiple rectangles. It can do that with an array list of `Point` or

`Rectangle` objects. The `paintComponent` method needs to draw them all. Replace the `moveRectangleTo` method with an `addRectangleAt` method that adds a rectangle at a given (x, y) position.

WORKED EXAMPLE 11.1

Programming a Working Calculator



Problem Statement Implement arithmetic and scientific operations for a calculator. Use the sample program from Section 11.1 as a starting point.

Arithmetic

In the calculator program of Section 11.1, the buttons for the arithmetic operations didn't do any work. It is actually a bit subtle to implement the behavior of a calculator. Imagine the user who has just entered $3 +$. At this point, we can't yet perform the addition because we don't have the second operand. We need to store the value (3) and the operator (+) and keep on going. Now the user continues:

$3 + 4 *$

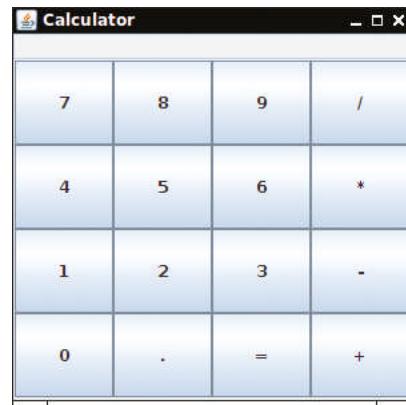
As soon as the * button is clicked, we can get to work and *add* 3 and 4. That is, we take the saved value and the newly entered value, and combine them with the *saved* operator. Then we save the * so that it can be executed later.

(Here, we implement a common household calculator in which multiplication and addition have the same precedence. In Chapter 15, you will learn how to implement a calculator in which multiplication has a higher precedence, as it does in mathematics.)

There is another subtlety, concerning the update of the calculator display. Consider the input

$1 3 + 4 * 2 =$

which arrives one button click at a time:



Button Clicked	Action	Display
1	Show 1 in display.	1
3	Add 3 to end of display.	13
+	Store 13 and + for later use.	13
4	Clear display, add 4.	4
*	Replace display with result of $13 + 4$. Store 17 and * for later use.	17
2	Clear display, add 2.	2
=	Replace display with result of $17 * 2$.	34

You may want to try this out with an actual calculator. Note the following:

- When an operator button is clicked and two operands are available, the display is updated with the result of the saved operation.

WE2 Chapter 11 Advanced User Interfaces

- The *first* digit button clicked after an operator clears the display. The other digit buttons append to the display. The display can't be cleared by the operator; it must be cleared by the first digit. (Otherwise, there would be no way for the user to see the result.)
- The = button puts the calculator into the same state as it was at the beginning, clearing the saved operation.

Now we have enough information to implement the arithmetic operator buttons. The calculator needs to remember

- the last value and operator.
- whether we are at the beginning or in the middle of entering a value.

We also need to remember the value that is currently being built up, but we can just take that from the display field.

```
public class CalculatorFrame extends JFrame
{
    private JTextField display;
    . .
    private double lastValue;
    private String lastOperator;
    private boolean startNewValue;

    public CalculatorFrame()
    {
        lastValue = 0;
        lastOperator = "=";
        startNewValue = true;
        . .
    }
    . .
}
```

The actionPerformed method of the digit button listeners appends the digit to the display; however, the display is cleared first if this was the first digit after an operator:

```
public void actionPerformed(ActionEvent event)
{
    if (startNewValue)
    {
        display.setText("");
        startNewValue = false;
    }
    display.setText(display.getText() + digit);
}
```

How does the method know which digit to use? It is passed to the constructor of the listener:

```
class DigitButtonListener implements ActionListener
{
    private String digit;

    public DigitButtonListener(String aDigit)
    {
        digit = aDigit;
    }
    . .
}
```

We construct digit buttons with this helper method:

```
public JButton makeDigitButton(String digit)
{
```

```

        JButton button = new JButton(digit);
        ActionListener listener = new DigitButtonListener(digit);
        button.addActionListener(listener);
        return button;
    }
}

```

The helper method is called for each digit button:

```

private void createButtonPanel()
{
    JPanel buttonPanel = new JPanel();
    buttonPanel.setLayout(new GridLayout(4, 4));

    buttonPanel.add(makeDigitButton("7"));
    buttonPanel.add(makeDigitButton("8"));
    buttonPanel.add(makeDigitButton("9"));
    . . .
}

```

We use the same strategy to pass the operator symbol to the `OperatorButtonListener`. Here is its `actionPerformed` method:

```

public void actionPerformed(ActionEvent event)
{
    if (!startNewValue)
    {
        double value = Double.parseDouble(display.getText());
        lastValue = calculate(lastValue, value, lastOperator);
        display.setText("") + lastValue);
        startNewValue = true;
    }

    lastOperator = operator;
}

```

First, we check whether the operator follows a value. If a user clicked two operators in a row, as in $3 + * 4$, we assume that the intent was to replace an incorrectly entered operator.

In the normal case, we combine the last value with the display value, using the *last* operator. We update the display with the result, and get ready to receive the next value.

We also store the current operator so that it can be evaluated later.

The `calculate` method simply combines its inputs:

```

public double calculate(double value1, double value2, String op)
{
    if (op.equals("+"))
    {
        return value1 + value2;
    }
    else if (op.equals("-"))
    {
        return value1 - value2;
    }
    else if (op.equals("*"))
    {
        return value1 * value2;
    }
    else if (op.equals("/"))
    {
        return value1 / value2;
    }
    else // "="
    {

```

```

        return value2;
    }
}

```

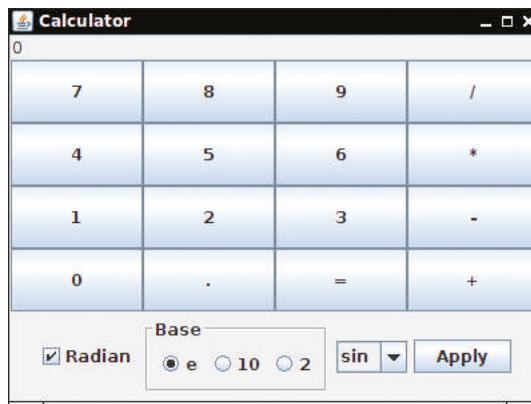
To understand the behavior for the = operator, think through an input $3 + 4 =$ followed by $5 * 6$. When the = button is clicked, the last operator (+) is executed, and = becomes the last operator. When the * button is clicked, the calculate method receives the last value (7), the display value (5), and the last operator (=). It should simply return the second operand (5), which will later be combined with the 6.

This completes the implementation of the arithmetic operators.

Mathematical Functions

In order to practice working with user-interface components, we will enhance the calculator with a few mathematical functions. The trigonometric functions sin, cos, and tan take an argument that can be interpreted as radians or degrees. We provide a check box to select radians. (Perhaps two radio buttons for radians and degrees would be clearer, but we want to practice using a checkbox). For the log and exp functions, we provide radio buttons to select one of three bases: e, 10, and 2. We place the functions into a combo box.

Clicking the Apply button applies the selected function with the selected options.



First, we need to set up the user interface. We need

- A checkbox for radians
- Three radio buttons
- A button group for the radio buttons
- A border for the radio buttons
- A combo box for the functions
- An Apply button

Let's get the radio buttons out of the way first:

```

private JPanel createBaseButtons()
{
    baseeButton = new JRadioButton("e");
    base10Button = new JRadioButton("10");
    base2Button = new JRadioButton("2");

    baseeButton.setSelected(true);

    ButtonGroup group = new ButtonGroup();
    group.add(baseeButton);

```

```

        group.add(base10Button);
        group.add(base2Button);

        JPanel basePanel = new JPanel();
        basePanel.add(baseeButton);
        basePanel.add(base10Button);
        basePanel.add(base2Button);
        basePanel.setBorder(new TitledBorder(new EtchedBorder(), "Base"));

        return basePanel;
    }
}

```

Here we create three radio buttons, select one of them, and add them to a button group. Note that the buttons are instance variables—we need to query their state later. However, the button group is only used by the Swing library, not our program. Therefore, it can be a local variable.

Finally, we add the buttons into a panel so that we can apply a border.

The remainder of the user interface is simpler. We just need to add the checkbox, combo box, radio buttons, and Apply button to a panel, then add that panel to the southern area of the frame's border layout.

```

private void createControlPanel()
{
    radianCheckBox = new JCheckBox("Radian");
    radianCheckBox.setSelected(true);

    mathOpCombo = new JComboBox();
    mathOpCombo.addItem("sin");
    mathOpCombo.addItem("cos");
    mathOpCombo.addItem("tan");
    mathOpCombo.addItem("log");
    mathOpCombo.addItem("exp");

    mathOpButton = new JButton("Apply");
    mathOpButton.addActionListener(new MathOpListener());

    JPanel controlPanel = new JPanel();
    controlPanel.add(radianCheckBox);
    controlPanel.add(createBaseButtons());
    controlPanel.add(mathOpCombo);
    controlPanel.add(mathOpButton);

    add(controlPanel, BorderLayout.SOUTH);
}

```

The only button that receives a listener is the Apply button. The other components change their state when they are clicked, and the listener of the Apply button reads the state when it calls the selected function.

Here is the listener code:

```

class MathOpListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double value = Double.parseDouble(display.getText());
        String mathOp = (String) mathOpCombo.getSelectedItem();

        double base = 10;
        if (baseeButton.isSelected()) { base = Math.E; }
        else if (base2Button.isSelected()) { base = 2; }

        boolean radian = radianCheckBox.isSelected();

```

WE6 Chapter 11 Advanced User Interfaces

```
        if (!radian && (mathOp.equals("sin")
            || mathOp.equals("cos") || mathOp.equals("tan")))
        {
            value = Math.toRadians(value);
        }

        if (mathOp.equals("sin"))
        {
            value = Math.sin(value);
        }
        else if (mathOp.equals("cos"))
        {
            value = Math.cos(value);
        }
        else if (mathOp.equals("tan"))
        {
            value = Math.tan(value);
        }
        else if (mathOp.equals("log"))
        {
            value = Math.log(value) / Math.log(base);
        }
        else if (mathOp.equals("exp"))
        {
            value = Math.pow(base, value);
        }
        display.setText("") + value);

        startnewValue = true;
    }
}
```

First, we get the function's argument from the display, and the base from the radio buttons. If we need to call a trigonometric function with degrees, we convert the argument to radians. That is what the Java library expects.

Then we execute the selected function and update the display. Finally, we set the `startnewValue` flag. If the user clicks a digit button, the display is cleared and the button becomes the first digit of a new value.

worked_example_1/CalculatorViewer.java

```
1 import javax.swing.JFrame;
2
3 public class CalculatorViewer
4 {
5     public static void main(String[] args)
6     {
7         JFrame frame = new CalculatorFrame();
8         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9         frame.setTitle("Calculator");
10        frame.setVisible(true);
11    }
12 }
```

worked_example_1/CalculatorFrame.java

```
1 import java.awt.BorderLayout;
2 import java.awt.GridLayout;
3 import java.awt.event.ActionListener;
```

```

4 import java.awt.event.ActionEvent;
5 import javax.swing.ButtonGroup;
6 import javax.swing.JButton;
7 import javax.swing.JCheckBox;
8 import javax.swing.JComboBox;
9 import javax.swing.JFrame;
10 import javax.swing.JPanel;
11 import javax.swing.JRadioButton;
12 import javax.swing.JTextField;
13 import javax.swing.border.EtchedBorder;
14 import javax.swing.border.TitledBorder;
15
16 /**
17      This frame contains a panel that displays buttons
18      for a calculator and a panel with a text field to
19      specify the result of calculation.
20 */
21 public class CalculatorFrame extends JFrame
22 {
23     private JTextField display;
24     private JCheckBox radianCheckBox;
25     private JRadioButton baseeButton;
26     private JRadioButton base10Button;
27     private JRadioButton base2Button;
28     private JComboBox mathOpCombo;
29     private JButton mathOpButton;
30
31     private double lastValue;
32     private String lastOperator;
33     private boolean startNewValue;
34
35     private static final int FRAME_WIDTH = 400;
36     private static final int FRAME_HEIGHT = 300;
37
38     public CalculatorFrame()
39     {
40         createButtonPanel();
41         createControlPanel();
42
43         display = new JTextField("0");
44         display.setEditable(false);
45         add(display, BorderLayout.NORTH);
46
47         lastValue = 0;
48         lastOperator = "=";
49         startNewValue = true;
50
51         setSize(FRAME_WIDTH, FRAME_HEIGHT);
52     }
53
54 /**
55      Creates the control panel with the text field
56      and buttons on the frame.
57 */
58     private void createButtonPanel()
59     {
60         JPanel buttonPanel = new JPanel();
61         buttonPanel.setLayout(new GridLayout(4, 4));
62

```

WE8 Chapter 11 Advanced User Interfaces

```
63     buttonPanel.add(makeDigitButton("7"));
64     buttonPanel.add(makeDigitButton("8"));
65     buttonPanel.add(makeDigitButton("9"));
66     buttonPanel.add(makeOperatorButton("/"));
67     buttonPanel.add(makeDigitButton("4"));
68     buttonPanel.add(makeDigitButton("5"));
69     buttonPanel.add(makeDigitButton("6"));
70     buttonPanel.add(makeOperatorButton("*"));
71     buttonPanel.add(makeDigitButton("1"));
72     buttonPanel.add(makeDigitButton("2"));
73     buttonPanel.add(makeDigitButton("3"));
74     buttonPanel.add(makeOperatorButton("-"));
75     buttonPanel.add(makeDigitButton("0"));
76     buttonPanel.add(makeDigitButton("."));
77     buttonPanel.add(makeOperatorButton("="));
78     buttonPanel.add(makeOperatorButton("+"));
79
80     add(buttonPanel, BorderLayout.CENTER);
81 }
82
83 class MathOpListener implements ActionListener
84 {
85     public void actionPerformed(ActionEvent event)
86     {
87         double value = Double.parseDouble(display.getText());
88         String mathOp = (String) mathOpCombo.getSelectedItem();
89
90         double base = 10;
91         if (baseeButton.isSelected()) { base = Math.E; }
92         else if (base2Button.isSelected()) { base = 2; }
93
94         boolean radian = radianCheckBox.isSelected();
95         if (!radian && (mathOp.equals("sin")
96                         || mathOp.equals("cos") || mathOp.equals("tan")))
97         {
98             value = Math.toRadians(value);
99         }
100
101        if (mathOp.equals("sin"))
102        {
103            value = Math.sin(value);
104        }
105        else if (mathOp.equals("cos"))
106        {
107            value = Math.cos(value);
108        }
109        else if (mathOp.equals("tan"))
110        {
111            value = Math.tan(value);
112        }
113        else if (mathOp.equals("log"))
114        {
115            value = Math.log(value) / Math.log(base);
116        }
117        else if (mathOp.equals("exp"))
118        {
119            value = Math.pow(base, value);
120        }
121        display.setText("") + value);
122 }
```

```

123         startnewValue = true;
124     }
125 }
126
127 private JPanel createBaseButtons()
128 {
129     baseeButton = new JRadioButton("e");
130     base10Button = new JRadioButton("10");
131     base2Button = new JRadioButton("2");
132
133     baseeButton.setSelected(true);
134
135     ButtonGroup group = new ButtonGroup();
136     group.add(baseeButton);
137     group.add(base10Button);
138     group.add(base2Button);
139
140     JPanel basePanel = new JPanel();
141     basePanel.add(baseeButton);
142     basePanel.add(base10Button);
143     basePanel.add(base2Button);
144     basePanel.setBorder(new TitledBorder(new EtchedBorder(), "Base"));
145
146     return basePanel;
147 }
148
149 private void createControlPanel()
150 {
151     radianCheckBox = new JCheckBox("Radian");
152     radianCheckBox.setSelected(true);
153
154     mathOpCombo = new JComboBox();
155     mathOpCombo.addItem("sin");
156     mathOpCombo.addItem("cos");
157     mathOpCombo.addItem("tan");
158     mathOpCombo.addItem("log");
159     mathOpCombo.addItem("exp");
160
161     mathOpButton = new JButton("Apply");
162     mathOpButton.addActionListener(new MathOpListener());
163
164     JPanel controlPanel = new JPanel();
165     controlPanel.add(radianCheckBox);
166     controlPanel.add(createBaseButtons());
167     controlPanel.add(mathOpCombo);
168     controlPanel.add(mathOpButton);
169
170     add(controlPanel, BorderLayout.SOUTH);
171 }
172
173 /**
174  * Combines two values with an operator.
175  * @param value1 the first value
176  * @param value2 the second value
177  * @param op an operator (+, -, *, /, or =)
178 */
179 public double calculate(double value1, double value2, String op)
180 {
181     if (op.equals("+"))
182     {

```

```

183         return value1 + value2;
184     }
185     else if (op.equals("-"))
186     {
187         return value1 - value2;
188     }
189     else if (op.equals("*"))
190     {
191         return value1 * value2;
192     }
193     else if (op.equals("/"))
194     {
195         return value1 / value2;
196     }
197     else // "="
198     {
199         return value2;
200     }
201 }
202
203 class DigitButtonListener implements ActionListener
204 {
205     private String digit;
206
207     /**
208      Constructs a listener whose actionPerformed method adds a digit
209      to the display.
210      @param aDigit the digit to add
211     */
212     public DigitButtonListener(String aDigit)
213     {
214         digit = aDigit;
215     }
216
217     public void actionPerformed(ActionEvent event)
218     {
219         if (startNewValue)
220         {
221             display.setText("");
222             startNewValue = false;
223         }
224         display.setText(display.getText() + digit);
225     }
226 }
227
228 /**
229  Makes a button representing a digit of a calculator.
230  @param digit the digit of the calculator
231  @return the button of the calculator
232 */
233 JButton makeDigitButton(String digit)
234 {
235     JButton button = new JButton(digit);
236     ActionListener listener = new DigitButtonListener(digit);
237     button.addActionListener(listener);
238     return button;
239 }

```

```
240
241 class OperatorButtonListener implements ActionListener
242 {
243     private String operator;
244
245     /**
246      Constructs a listener whose actionPerformed method
247      schedules an operator for execution.
248     */
249     public OperatorButtonListener(String anOperator)
250     {
251         operator = anOperator;
252     }
253
254     public void actionPerformed(ActionEvent event)
255     {
256         if (!startNewValue)
257         {
258             double value = Double.parseDouble(display.getText());
259             lastValue = calculate(lastValue, value, lastOperator);
260             display.setText(" " + lastValue);
261             startNewValue = true;
262         }
263
264         lastOperator = operator;
265     }
266 }
267
268 /**
269  * Makes a button representing an operator of a calculator.
270  * @param op the operator of the calculator
271  * @return button the button of the calculator
272 */
273 JButton makeOperatorButton(String op)
274 {
275     JButton button = new JButton(op);
276     ActionListener listener = new OperatorButtonListener(op);
277     button.addActionListener(listener);
278     return button;
279 }
280 }
```

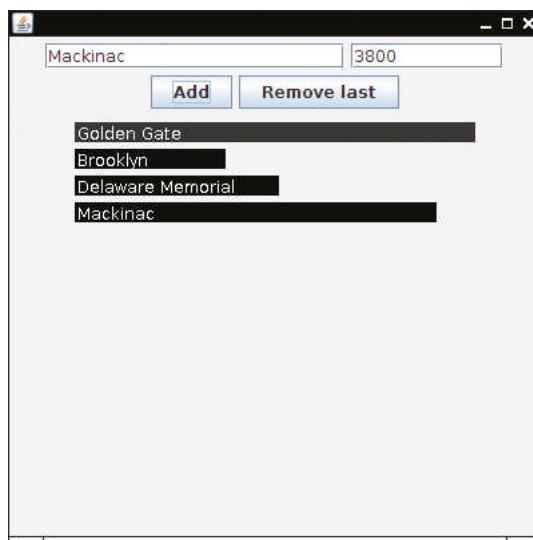

WORKED EXAMPLE 11.2

Adding Mouse and Keyboard Support to the Bar Chart Creator

Problem Statement Enhance the bar chart creator of Worked Example 10.1 and add support for mouse and keyboard operations as follows:

- If the user selects one of the bars by clicking on it, the selected bar is displayed in red.
- If the user clicks inside the row of the selected bar again, the bar is resized so that it extends to the location of the mouse click.
- If the user clicks on a different row, the bar is selected. However, if the user clicks below the last bar, a new bar (with a blank label) is added to the chart.
- Users can also use the arrow keys. The left and right arrow keys resize the selected bar. The up and down arrow keys move the selected bar up or down.

You may want to run the program to get a feel for these operations. You have to pay close attention when you execute the keyboard commands. The chart must have *focus* in order to receive them. The application is made up of five components: two text fields, two buttons, and the chart. When the application first starts, the first text field has focus. That is, if you type keys on the keyboard, they are directed to the first text field. Try it out: Type a few letters, then the left and right arrow keys. The arrow keys move the cursor. Now press the Tab key. The focus is transferred to the second text field. Type a few numbers, then the left and right arrow keys. The arrow keys move the cursor in the second text field. (If you look carefully, you will notice that there is no longer a cursor in the first one.) Press Tab again. The focus is now on the first button. A button with focus is marked in some way that depends on the “look and feel” of your Java installation. For example, here the button with focus has a rectangle around the button text.



Press the space bar. That's the same as clicking the button. A bar is added to the chart. Now press Tab again to shift focus to the next button. Then press Tab one more time. Now the focus is on the chart, and you can use the arrow keys to adjust the bar widths.

This may seem like a tedious way of navigating a user interface. However, there are many people with a handicap that prevents them from using a mouse effectively. These people rely on keyboard navigation. The Swing library provides the focus mechanism—you need not worry about handling Tab key presses. In this worked example, we add the keyboard

shortcuts to the chart so that all users have a convenient way of editing the chart, either with the commands or the keyboard.

Updating the ChartFrame Class

To implement these enhancements, we first modify the frame class and add mouse and key listeners to the chart component. (See Special Topic 11.1 for key listeners.)

```
public class ChartFrame extends JFrame
{
    .
    .
    class ChartMouseListener implements MouseListener
    {
        public void mousePressed(MouseEvent event)
        {
            int x = event.getX();
            int y = event.getY();
            Process mouse press at (x,y).
        }
        // Do-nothing methods
        public void mouseReleased(MouseEvent event) {}
        public void mouseClicked(MouseEvent event) {}
        public void mouseEntered(MouseEvent event) {}
        public void mouseExited(MouseEvent event) {}
    }

    class ChartKeyListener implements KeyListener
    {
        public void keyPressed(KeyEvent event)
        {
            String key = KeyStroke.getKeyStrokeForEvent(event).toString();
            key = key.replace("pressed ", "");
            Process key press.
        }
        // Do-nothing methods
        public void keyReleased(KeyEvent event) {}
        public void keyTyped(KeyEvent event) {}
    }

    public ChartFrame()
    {
        chart = new ChartComponent();
        chart.addMouseListener(new ChartMouseListener());
        chart.addKeyListener(new ChartKeyListener());
        chart.setFocusable(true);
        .
        .
    }
    .
}
```

Note the call to the `setFocusable` method. By default, a plain Swing component is not focusable. We want ours to be focusable, so that it can receive keystrokes.

In the key listener, we invoke an action for each key:

```
public void keyPressed(KeyEvent event)
{
    .
    .
    if (key.equals("RIGHT")) { chart.growSelected(1); }
    else if (key.equals("LEFT")) { chart.growSelected(-1); }
    else if (key.equals("DOWN")) { chart.changeSelected(1); }
    else if (key.equals("UP")) { chart.changeSelected(-1); }
```

```
    . . .
}
```

The `growSelected` method will grow or shrink the selected bar by the given amount. The `changeSelected` method will change the selection to the next or previous bar. We will implement these methods in the next section.

Unfortunately, it isn't so easy for the frame to find out what a mouse click should do. Should the mouse click select or resize a bar? That depends on the location of the click. Does it fall on the currently selected bar? The frame class doesn't know the exact geometry of the chart, so it can't make that decision. Therefore, the mouse handler simply passes the mouse coordinates to the chart. (We will add a `click` method to the chart class in the next section.)

```
public void mousePressed(MouseEvent event)
{
    . . .
    chart.click(x, y);
    chart.requestFocus();
}
```

We also add a call to the `requestFocus` method. This way, the chart gets focus when you click on it. Try it out: After clicking on the chart, the arrow keys change the chart—you don't have to use the Tab key.

Updating the ChartComponent and Bar Classes

The `ChartComponent` class must remember the selected bar. In the `paintComponent` method, we must paint the selected bar in a special way. This is achieved by passing a Boolean flag to the bar's `draw` method that is true if the bar is the selected one:

```
public class ChartComponent extends JPanel
{
    private ArrayList<Bar> bars;
    private double maxValue;
    private int selected;
    . . .
    public void paintComponent(Graphics g)
    {
        int y = GAP;
        double scale = getWidth() / maxValue;
        for (int i = 0; i < bars.size(); i++)
        {
            Bar b = bars.get(i);
            b.draw(g, y, scale, i == selected);
            y = y + Bar.HEIGHT + GAP;
        }
    }
}
```

In the `draw` method, a selected bar is painted in red:

```
public class Bar
{
    . . .
    public void draw(Graphics g, int y, double scale, boolean selected)
    {
        final int GAP = 2;
        if (selected)
        {
            g.setColor(Color.RED);
        }
        g.fillRect(0, y, (int) (value * scale), HEIGHT);
        g.setColor(Color.WHITE);
```

```

        g.drawString(label, GAP, y + HEIGHT - GAP);
        g.setColor(Color.BLACK);
    }
}

```

When the selected bar is changed with the keyboard, the component's `changeSelected` method is called. It simply updates the value of the selected variable, wrapping around (from the last bar to the first or the first to the last) if necessary. Here it is:

```

/**
 * Change the selected bar.
 * @param increment +1 or -1 to select the next or previous bar
 */
public void changeSelected(int increment)
{
    if (bars.size == 0) { return; }
    selected = selected + increment;
    // Wrap around at the ends
    int n = bars.size();
    if (selected >= n) { selected = selected - n; }
    if (selected < 0) { selected = selected + n; }
    repaint();
}

```

Note the call to `repaint` at the end. When a different bar is selected, the chart must be repainted to show the new selection.

The `growSelected` method grows or shrinks the selected bar. We change the bar's value by the equivalent of one pixel, converted to the units used by the values. Note that we needed to add `getValue` and `setValue` methods to the `Bar` class. These were not required in the original version, which had no user interface for modifying a bar's value.

```

public void growSelected(int increment)
{
    if (bars.size == 0) { return; }
    Bar b = bars.get(selected);
    double value = b.getValue();
    value = value + increment * maxValue / getWidth();
    b.setValue(value);
    repaint();
}

```

Finally, we need to tackle the `click` method. When the mouse clicks at (x, y) , we first need to determine which bar is selected. Bars correspond to the y -value. As you can see from the `paintComponent` method, each bar occupies `GAP + Bar.HEIGHT` pixels. To convert the pixels to a bar index, we must divide by this value.

```

public void click(int x, int y)
{
    int row = y / (GAP + Bar.HEIGHT);
    . .
}

```

If the row is the currently selected one, we update the value of the bar. Again, we need to convert from pixels to bar units.

```

double value = x * maxValue / getWidth();

if (row == selected)
{
    bars.get(row).setValue(value);
}

```

Otherwise, if the row is larger than the number of bars, we add a new bar:

```
else if (row >= bars.size())
{
    bars.add(new Bar("", value));
    selected = bars.size() - 1;
}
```

Otherwise, we simply select a new bar. Again, we need to repaint at the end of the method.

```
else
{
    selected = row;
}
repaint();
```

That completes the `click` method and the implementation of the program.

Again, it is instructive to consider the division of labor between the component and frame classes. The component class knows how to paint itself, and it knows about the current selection state. The frame class has no knowledge of these details. It simply collects the user input and passes it to the component.

worked_example_2/ChartViewer.java

```
1 import javax.swing.JFrame;
2
3 /**
4  * This program displays an editable bar chart.
5 */
6 public class ChartViewer
7 {
8     public static void main(String[] args)
9     {
10         JFrame frame = new ChartFrame();
11         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12         frame.setVisible(true);
13     }
14 }
```

worked_example_2/ChartFrame.java

```
1 import java.awt.Dimension;
2 import java.awt.event.ActionEvent;
3 import java.awt.event.ActionListener;
4 import java.awt.event.KeyEvent;
5 import java.awt.event.KeyListener;
6 import java.awt.event.MouseEvent;
7 import java.awt.event.MouseListener;
8 import javax.swing.KeyStroke;
9 import javax.swing.JButton;
10 import javax.swing.JFrame;
11 import javax.swing.JLabel;
12 import javax.swing.JPanel;
13 import javax.swing.JTextField;
14
15 /**
16  * A frame that allows users to edit a bar chart.
17 */
18 public class ChartFrame extends JFrame
19 {
```

```

20     private static final int FRAME_WIDTH = 400;
21     private static final int FRAME_HEIGHT = 400;
22
23     private static final int CHART_WIDTH = 300;
24     private static final int CHART_HEIGHT = 300;
25
26     private static final String DEFAULT_LABEL = "Description";
27     private static final double DEFAULT_VALUE = 100;
28
29     private JTextField labelField;
30     private JTextField valueField;
31     private JButton addButton;
32     private JButton removeButton;
33     private ChartComponent chart;
34
35     class ChartMouseListener implements MouseListener
36     {
37         public void mousePressed(MouseEvent event)
38         {
39             int x = event.getX();
40             int y = event.getY();
41             chart.click(x, y);
42             chart.requestFocus();
43         }
44         // Do-nothing methods
45         public void mouseReleased(MouseEvent event) {}
46         public void mouseClicked(MouseEvent event) {}
47         public void mouseEntered(MouseEvent event) {}
48         public void mouseExited(MouseEvent event) {}
49     }
50
51     class ChartKeyListener implements KeyListener
52     {
53         public void keyPressed(KeyEvent event)
54         {
55             String key = KeyStroke.getKeyStrokeForEvent(event).toString();
56             key = key.replace("pressed ", "");
57             if (key.equals("RIGHT")) { chart.growSelected(1); }
58             else if (key.equals("LEFT")) { chart.growSelected(-1); }
59             else if (key.equals("DOWN")) { chart.changeSelected(1); }
60             else if (key.equals("UP")) { chart.changeSelected(-1); }
61         }
62         // Do-nothing methods
63         public void keyReleased(KeyEvent event) {}
64         public void keyTyped(KeyEvent event) {}
65     }
66
67     public ChartFrame()
68     {
69         chart = new ChartComponent();
70         chart.setPreferredSize(new Dimension(CHART_WIDTH, CHART_HEIGHT));
71         chart.append(DEFAULT_LABEL, DEFAULT_VALUE);
72         chart.addMouseListener(new ChartMouseListener());
73         chart.addKeyListener(new ChartKeyListener());
74         chart.setFocusable(true);
75
76         createTextFields();
77         createButtons();
78         createPanel();
79     }

```

```

80         setSize(FRAME_WIDTH, FRAME_HEIGHT);
81     }
82
83     private void createTextFields()
84     {
85         final int LABEL_FIELD_WIDTH = 20;
86         labelField = new JTextField(LABEL_FIELD_WIDTH);
87         labelField.setText(DEFAULT_LABEL);
88         final int VALUE_FIELD_WIDTH = 10;
89         valueField = new JTextField(VALUE_FIELD_WIDTH);
90         valueField.setText("'" + DEFAULT_VALUE);
91     }
92
93     class AddBarListener implements ActionListener
94     {
95         public void actionPerformed(ActionEvent event)
96         {
97             String label = labelField.getText();
98             double value = Double.parseDouble(valueField.getText());
99             chart.append(label, value);
100        }
101    }
102
103    class RemoveBarListener implements ActionListener
104    {
105        public void actionPerformed(ActionEvent event)
106        {
107            chart.removeSelected();
108        }
109    }
110
111    private void createButtons()
112    {
113        addButton = new JButton("Add");
114        addButton.addActionListener(new AddBarListener());
115        removeButton = new JButton("Remove last");
116        removeButton.addActionListener(new RemoveBarListener());
117    }
118
119    private void createPanel()
120    {
121        JPanel panel = new JPanel();
122        panel.add(labelField);
123        panel.add(valueField);
124        panel.add(addButton);
125        panel.add(removeButton);
126        panel.add(chart);
127        add(panel);
128    }
129}

```

worked_example_2/ChartComponent.java

```

1 import java.awt.Color;
2 import java.awt.Graphics;
3 import java.util.ArrayList;
4 import javax.swing.JComponent;
5

```

```
6  /**
7   * A component that draws a chart.
8  */
9 public class ChartComponent extends JComponent
10 {
11     private ArrayList<Bar> bars;
12     private double maxValue;
13     private int selected;
14
15     private static final int GAP = 5;
16
17     /**
18      Constructs an empty chart.
19     */
20     public ChartComponent()
21     {
22         bars = new ArrayList<Bar>();
23         maxValue = 1;
24         selected = 0;
25     }
26
27     /**
28      Appends a bar to this chart.
29      @param label the label for the bar
30      @param value the value of the bar
31     */
32     public void append(String label, double value)
33     {
34         bars.add(new Bar(label, value));
35         selected = bars.size() - 1;
36         if (value > maxValue) { maxValue = value; }
37         repaint();
38     }
39
40     /**
41      Process a mouse press at a given location.
42      @param x the x-coordinate of the mouse press
43      @param y the y-coordinate of the mouse press
44     */
45     public void click(int x, int y)
46     {
47         int row = y / (GAP + Bar.HEIGHT);
48         double value = x * maxValue / getWidth();
49
50         if (row == selected)
51         {
52             bars.get(row).setValue(value);
53         }
54         else if (row >= bars.size())
55         {
56             bars.add(new Bar("", value));
57             selected = bars.size() - 1;
58         }
59         else
60         {
61             selected = row;
62         }
63         repaint();
64     }
65 }
```

```

66  /**
67   * Grows or shrinks the selected bar.
68   * @param increment the amount (in pixels) by which to adjust the bar
69   */
70  public void growSelected(int increment)
71  {
72      if (bars.size() == 0) { return; }
73      Bar b = bars.get(selected);
74      double value = b.getValue();
75      value = value + increment * maxValue / getWidth();
76      b.setValue(value);
77      repaint();
78  }
79
80 /**
81  * Change the selected bar.
82  * @param increment +1 or -1 to select the next or previous bar
83  */
84  public void changeSelected(int increment)
85  {
86      if (bars.size() == 0) { return; }
87      selected = selected + increment;
88      // Wrap around at the ends
89      int n = bars.size();
90      if (selected >= n) { selected = selected - n; }
91      if (selected < 0) { selected = selected + n; }
92      repaint();
93  }
94
95 /**
96  * Removes the currently selected bar.
97  */
98  public void removeSelected()
99  {
100     if (bars.size() == 0) { return; }
101     bars.remove(selected);
102     if (selected == bars.size())
103     {
104         selected--;
105     }
106     repaint();
107 }
108
109 public void paintComponent(Graphics g)
110 {
111     int y = GAP;
112     double scale = getWidth() / maxValue;
113     for (int i = 0; i < bars.size(); i++)
114     {
115         Bar b = bars.get(i);
116         b.draw(g, y, scale, i == selected);
117         y = y + Bar.HEIGHT + GAP;
118     }
119 }
120 }
```

worked_example_2/Bar.java

```

1 import java.awt.Color;
2 import java.awt.Graphics;
```

```
3  /**
4   * A bar of a bar chart.
5   */
6  public class Bar
7  {
8    private String label;
9    private double value;
10
11   public static final int HEIGHT = 15;
12
13  /**
14   * Constructs a bar with a given label and value.
15   * @param aLabel the label for this bar
16   * @param aValue the value of this bar
17   */
18  public Bar(String aLabel, double aValue)
19  {
20    label = aLabel;
21    value = aValue;
22  }
23
24  public void setValue(double newValue)
25  {
26    value = newValue;
27  }
28
29  public double getValue()
30  {
31    return value;
32  }
33
34  /**
35   * Draws this bar.
36   * @param g the Graphics object
37   * @param y the top of the bar
38   * @param scale the horizontal scale factor for fitting the bar into the component
39   * @param selected true if this bar is selected
40   */
41  public void draw(Graphics g, int y, double scale, boolean selected)
42  {
43    final int GAP = 2;
44    if (selected)
45    {
46      g.setColor(Color.RED);
47    }
48    g.fillRect(0, y, (int) (value * scale), HEIGHT);
49    g.setColor(Color.WHITE);
50    g.drawString(label, GAP, y + HEIGHT - GAP);
51    g.setColor(Color.BLACK);
52
53  }
54
55 }
```

OBJECT-ORIENTED DESIGN

CHAPTER GOALS

- To learn how to discover new classes and methods
- To use CRC cards for class discovery
- To identify inheritance, aggregation, and dependency relationships between classes
- To describe class relationships using UML class diagrams
- To apply object-oriented design techniques to building complex programs

CHAPTER CONTENTS

12.1 CLASSES AND THEIR RESPONSIBILITIES 578

12.2 RELATIONSHIPS BETWEEN CLASSES 582

HT1 Using CRC Cards and UML Diagrams in Program Design 586

ST1 Attributes and Methods in UML Diagrams 586

ST2 Multiplicities 587



© Petrea Alexandru/iStockphoto.

ST3 Aggregation, Association, and Composition 587

PT1 Make Parallel Arrays into Arrays of Objects 588

12.3 APPLICATION: PRINTING AN INVOICE 589

C&S Databases and Privacy 600

WE1 Simulating an Automatic Teller Machine 



© Petrea Alexandru/iStockphoto.

Successfully implementing a software system—as simple as your next homework project or as complex as the next air traffic monitoring system—requires a great deal of planning and design. In fact, for larger projects, the amount of time spent on planning and design is much greater than the amount of time spent on programming and testing.

Do you find that most of your homework time is spent in front of the computer, keying in code and fixing bugs? If so, you can probably save time by focusing on a better design before you start coding. This chapter tells you how to approach the design of an object-oriented program in a systematic manner.

12.1 Classes and Their Responsibilities

When you design a program, you work from a *requirements specification*, a description of what your program should do. The designer's task is to discover structures that make it possible to implement the requirements in a computer program. In the following sections, we will examine the steps of the design process.

12.1.1 Discovering Classes

To discover classes, look for nouns in the problem description.

When you solve a problem using objects and classes, you need to determine the classes required for the implementation. You may be able to reuse existing classes, or you may need to implement new ones.

One simple approach for discovering classes and methods is to look for the nouns and verbs in the requirements specification. Often, *nouns* correspond to classes, and *verbs* correspond to methods.

For example, suppose your job is to print an invoice such as the one in Figure 1.

INVOICE			
Sam's Small Appliances 100 Main Street Anytown, CA 98765			
Item	Qty	Price	Total
Toaster	3	\$29.95	\$89.85
Hair Dryer	1	\$24.95	\$24.95
Car Vacuum	2	\$19.99	\$39.98

AMOUNT DUE: \$154.78

Figure 1
An Invoice

Concepts from the problem domain are good candidates for classes.

Obvious classes that come to mind are `Invoice`, `LineItem`, and `Customer`. It is a good idea to keep a list of *candidate classes* on a whiteboard or a sheet of paper. As you brainstorm, simply put all ideas for classes onto the list. You can always cross out the ones that weren't useful after all.

In general, concepts from the problem domain, be it science, business, or a game, often make good classes. Examples are

- `Cannonball`
- `CashRegister`
- `Monster`

The name for such a class should be a noun that describes the concept.

Not all classes can be discovered from the program requirements. Most complex programs need classes for tactical purposes, such as file or database access, user interfaces, control mechanisms, and so on.

Some of the classes that you need may already exist, either in the standard library or in a program that you developed previously. You also may be able to use inheritance to extend existing classes into classes that match your needs.

What might not be a good class? If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track. For example, your homework assignment might be to write a program that prints paychecks. Suppose you start by trying to design a class `PaycheckProgram`. What would an object of this class do? An object of this class would have to do everything that the homework needs to do. That doesn't simplify anything. A better class would be `Paycheck`. Then your program can manipulate one or more `Paycheck` objects.

Another common mistake, often made by students who are used to writing programs that consist of static methods, is to turn an action into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful methods of the `Paycheck` class, such as `computeTaxes`, that help you solve the assignment.

Finally, a common error is to overdo the class discovery process. For example, should an address be an object of an `Address` class, or should it simply be a string? There is no perfect answer—it depends on the task that you want to solve. If your software needs to analyze addresses (for example, to determine shipping costs), then

In a class scheduling system, potential classes from the problem domain include `Class`, `LectureHall`, `Instructor`, and `Student`.



© Oleg Prikhodko/Stockphoto.

an Address class is an appropriate design. However, if your software will never need such a capability, you should not waste time on an overly complex design. It is your job to find a balanced design; one that is neither too limiting nor excessively general.

12.1.2 The CRC Card Method

A CRC card describes a class, its responsibilities, and its collaborating classes.

Once you have identified a set of classes, you define the behavior for each class. Find out what methods you need to provide for each class in order to solve the programming problem. A simple rule for finding these methods is to look for *verbs* in the task description, then match the verbs to the appropriate objects. For example, in the invoice program, a class needs to compute the amount due. Now you need to figure out *which class* is responsible for this method. Do customers compute what they owe? Do invoices total up the amount due? Do the items total themselves up? The best choice is to make “compute amount due” the responsibility of the *Invoice* class.

An excellent way to carry out this task is the “**CRC card** method.” *CRC* stands for “classes”, “responsibilities”, “collaborators”, and in its simplest form, the method works as follows: Use an index card for each *class* (see Figure 2). As you think about verbs in the task description that indicate methods, you pick the card of the class that you think should be responsible, and write that *responsibility* on the card.

For each responsibility, you record which other classes are needed to fulfill it. Those classes are the **collaborators**.

For example, suppose you decide that an invoice should compute the amount due. Then you write “compute amount due” on the left-hand side of an index card with the title *Invoice*.

If a class can carry out that responsibility by itself, do nothing further. But if the class needs the help of other classes, write the names of these collaborators on the right-hand side of the card.

To compute the total, the invoice needs to ask each line item about its total price. Therefore, the *LineItem* class is a collaborator.

This is a good time to look up the index card for the *LineItem* class. Does it have a “get total price” method? If not, add one.

How do you know that you are on the right track? For each responsibility, ask yourself how it can actually be done, using the responsibilities written on the various cards. Many people find it helpful to group the cards on a table so that the

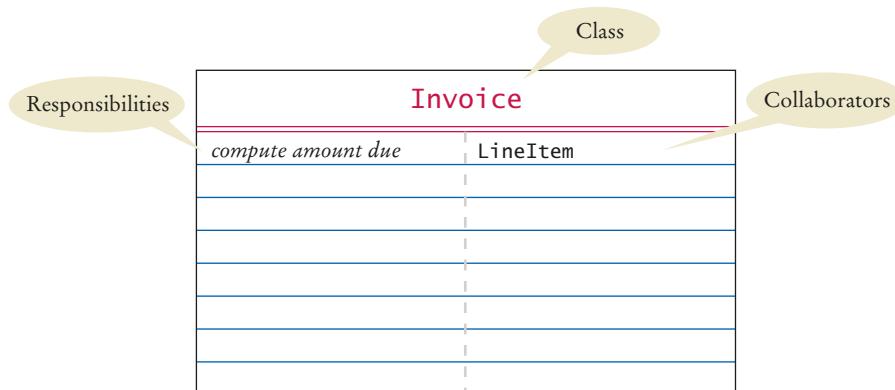


Figure 2 A CRC Card

collaborators are close to each other, and to simulate tasks by moving a token (such as a coin) from one card to the next to indicate which object is currently active.

Keep in mind that the responsibilities that you list on the CRC card are on a *high level*. Sometimes a single responsibility may need two or more Java methods for carrying it out. Some researchers say that a CRC card should have no more than three distinct responsibilities.

The CRC card method is informal on purpose, so that you can be creative and discover classes and their properties. Once you find that you have settled on a good set of classes, you will want to know how they are related to each other. Can you find classes with common properties, so that some responsibilities can be taken care of by a common superclass? Can you organize classes into clusters that are independent of each other? Finding class relationships and documenting them with diagrams is the topic of Section 12.2.

12.1.3 Cohesion

The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

A class should represent a single concept. The public methods and constants that the public interface exposes should be *cohesive*. That is, all interface features should be closely related to the single concept that the class represents.

If you find that the public interface of a class refers to multiple concepts, then that is a good sign that it may be time to use separate classes instead. Consider, for example, the public interface of a `CashRegister` class:

```
public class CashRegister
{
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    ...
    public void enterPayment(int dollars, int quarters,
                           int dimes, int nickels, int pennies) { . . . }
    ...
}
```

There are really two concepts here: a cash register that holds coins and computes their total, and the values of individual coins. (For simplicity, we assume that the cash register only holds coins, not bills.)

It makes sense to have a separate `Coin` class and have coins responsible for knowing their values.

```
public class Coin
{
    ...
    public Coin(double aValue, String aName) { . . . }
    public double getValue() { . . . }
    ...
}
```

Then the `CashRegister` class can be simplified:

```
public class CashRegister
{
    ...
    public void enterPayment(int coinCount, Coin coinType) { . . . }
    ...
}
```



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a sample program using the `Coin` and `CashRegister` classes.

Now the `CashRegister` class no longer needs to know anything about coin values. The same class can equally well handle euros or zorkmids!

This is clearly a better solution, because it separates the responsibilities of the cash register and the coins.



1. What is the rule of thumb for finding classes?
2. Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?
3. Suppose the invoice is to be saved to a file. Name a likely collaborator.
4. Looking at the invoice in Figure 1, what is a likely responsibility of the `Customer` class?
5. What do you do if a CRC card has ten responsibilities?

Practice It Now you can try these exercises at the end of the chapter: R12.4, R12.8.

12.2 Relationships Between Classes

When designing a program, it is useful to document the relationships between classes. This helps you in a number of ways. For example, if you find classes with common behavior, you can save effort by placing the common behavior into a superclass. If you know that some classes are *not* related to each other, you can assign different programmers to implement each of them, without worrying that one of them has to wait for the other.

In the following sections, we will describe the most common types of relationships.

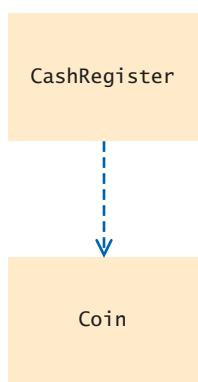
12.2.1 Dependency

A class depends on another class if it uses objects of that class.

Many classes need other classes in order to do their jobs. For example, in Section 12.1.3, we described a design of a `CashRegister` class that depends on the `Coin` class to determine the value of the payment.

The dependency relationship is sometimes nicknamed the “knows about” relationship. The cash register in Section 12.1.3 knows that there are coin objects. In contrast, the `Coin` class does *not* depend on the `CashRegister` class. Coins have no idea that they are being collected in cash registers, and they can carry out their work without ever calling any method in the `CashRegister` class.

To visualize relationships, such as dependency between classes, programmers draw class diagrams. In this book, we use the UML (“Unified Modeling Language”) notation for objects and classes. UML is a notation for object-oriented analysis and design invented



Too many dependencies make a system difficult to manage.

© visual7/iStockphoto.

Figure 3

Dependency Relationship Between the `CashRegister` and `Coin` Classes

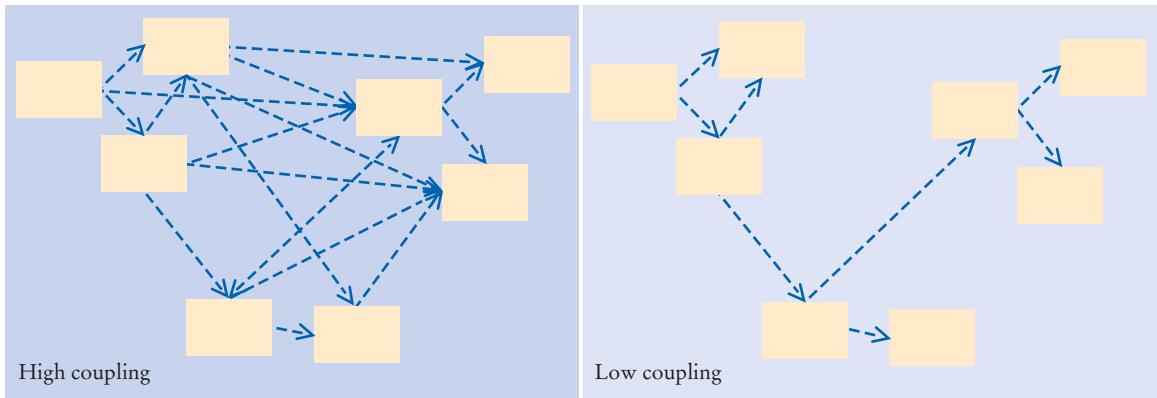


Figure 4 High and Low Coupling Between Classes

by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented software development. The UML notation distinguishes between *object diagrams* and *class diagrams*. An object diagram shows individual objects, their attributes, and the relationships between them. Chapter 8 has several object diagrams. A class diagram shows classes and the relationships between them. In Chapter 9, you saw class diagrams that show inheritance relationships. In the UML notation, we underline the names of classes in object diagrams but not in class diagrams.

In a class diagram, dependency is denoted by a dashed line with a $>$ -shaped open arrow tip. The arrow tip points to the class on which the other class depends. Figure 3 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

If many classes of a program depend on each other, then we say that the **coupling** between classes is high. Conversely, if there are few dependencies between classes, then we say that the coupling is low (see Figure 4).

Why does coupling matter? If the `Coin` class changes in the next release of the program, all the classes that depend on it may be affected. If the change is drastic, the coupled classes must all be updated. Furthermore, if we would like to use a class in another program, we have to take with it all the classes on which it depends. Thus, we want to remove unnecessary coupling between classes.

It is a good practice to minimize the coupling (i.e., dependency) between classes.

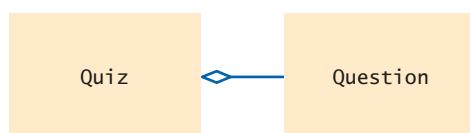
A class aggregates another if its objects contain objects of the other class.

12.2.2 Aggregation

Another fundamental relationship between classes is the “aggregation” relationship (which is informally known as the “has-a” relationship).

The **aggregation** relationship states that objects of one class contain objects of another class. Consider a quiz that is made up of questions. Because each quiz has one or more questions, we say that the class `Quiz` *aggregates* the class `Question`. In the UML notation, aggregation is denoted by a line with a diamond-shaped symbol attached to the aggregating class (see Figure 5).

Figure 5
Class Diagram
Showing Aggregation



*A car has a motor and tires.
In object-oriented design,
this “has-a” relationship
is called aggregation.*



© bojan fatur/istockphoto.

Finding out about aggregation is very helpful for deciding how to implement classes. For example, when you implement the Quiz class, you will want to store the questions of a quiz as an instance variable.

Because a quiz can have any number of questions, an array or array list is a good choice for collecting them:

```
public class Quiz
{
    private ArrayList<Question> questions;
    ...
}
```

Aggregation is a stronger form of dependency. If a class has objects of another class, it certainly knows about the other class. However, the converse is not true. For example, a class may use the Scanner class without ever declaring an instance variable of class Scanner. The class may simply construct a local variable of type Scanner, or its methods may receive Scanner objects as arguments. This use is not aggregation because the objects of the class don't contain Scanner objects—they just create or receive them for the duration of a single method.

Generally, you need aggregation when an object needs to remember another object *between method calls*.

12.2.3 Inheritance

Inheritance is a relationship between a more general class (the superclass) and a more specialized class (the subclass). This relationship is often described as the “*is-a*” relationship. Every truck *is a* vehicle. Every savings account *is a* bank account.

Inheritance is sometimes abused. For example, consider a Tire class that describes a car tire. Should the class Tire be a subclass of a class Circle? It sounds convenient. There are quite a few useful methods in the Circle class—for example, the Tire class may inherit methods that compute the radius, perimeter, and center point, which should come in handy when drawing tire shapes. Though it may be convenient for the programmer, this arrangement makes no sense conceptually. It isn't true that every tire is a circle. Tires are car parts, whereas circles are geometric objects. There is a relationship between tires and circles, though. A tire *has a* circle as its boundary. Use aggregation:

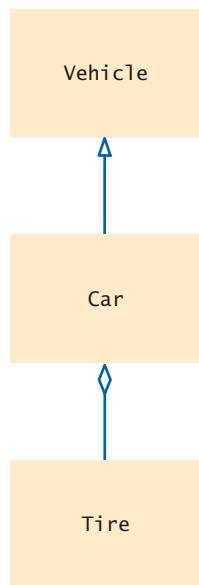
```
public class Tire
{
    private String rating;
    private Circle boundary;
    ...
}
```

Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.

FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download the complete Quiz and Question classes.

Figure 6
UML Notation for Inheritance and Aggregation



Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.

You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.

Here is another example: Every car *is a* vehicle. Every car *has a* tire (in fact, it typically has four or, if you count the spare, five). Thus, you would use inheritance from Vehicle and use aggregation of Tire objects (see Figure 6 for the UML diagram):

```

public class Car extends Vehicle
{
    private Tire[] tires;
    ...
}
  
```

The arrows in the UML notation can get confusing. Table 1 shows a summary of the four UML relationship symbols that we use in this book.

Table 1 UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance	—→	Solid	Triangle
Interface Implementation	- - - →	Dotted	Triangle
Aggregation	◊ —→	Solid	Diamond
Dependency	- - - →	Dotted	Open

SELF CHECK



6. Consider the CashRegisterTester class of Section 12.1.3. On which classes does it depend?
7. Consider the Question and ChoiceQuestion objects of Chapter 9. How are they related?
8. Consider the Quiz class described in Section 12.2.2. Suppose a quiz contains a mixture of Question and ChoiceQuestion objects. Which classes does the Quiz class depend on?
9. Why should coupling be minimized between classes?
10. In an e-mail system, messages are stored in a mailbox. Draw a UML diagram that shows the appropriate aggregation relationship.
11. You are implementing a system to manage a library, keeping track of which books are checked out by whom. Should the Book class aggregate Patron or the other way around?
12. In a library management system, what would be the relationship between classes Patron and Author?

Practice It

Now you can try these exercises at the end of the chapter: R12.5, R12.6, R12.10.

HOW TO 12.1**Using CRC Cards and UML Diagrams in Program Design**

Before writing code for a complex problem, you need to design a solution. The methodology introduced in this chapter suggests that you follow a design process that is composed of the following tasks:

- Discover classes.
- Determine the responsibilities of each class.
- Describe the relationships between the classes.

CRC cards and UML diagrams help you discover and record this information.

Step 1 Discover classes.

Highlight the nouns in the problem description. Make a list of the nouns. Cross out those that don't seem to be reasonable candidates for classes.

Step 2 Discover responsibilities.

Make a list of the major tasks that your system needs to fulfill. From those tasks, pick one that is not trivial and that is intuitive to you. Find a class that is responsible for carrying out that task. Make an index card and write the name and the task on it. Now ask yourself how an object of the class can carry out the task. It probably needs help from other objects. Then make CRC cards for the classes to which those objects belong and write the responsibilities on them.

Don't be afraid to cross out, move, split, or merge responsibilities. Rip up cards if they become too messy. This is an informal process.

You are done when you have walked through all major tasks and are satisfied that they can all be solved with the classes and responsibilities that you discovered.

Step 3 Describe relationships.

Make a class diagram that shows the relationships between all the classes that you discovered.

Start with inheritance—the *is-a* relationship between classes. Is any class a specialization of another? If so, draw inheritance arrows. Keep in mind that many designs, especially for simple programs, don't use inheritance extensively.

The “collaborators” column of the CRC card tells you which classes are used by that class. Draw dependency arrows for the collaborators on the CRC cards.

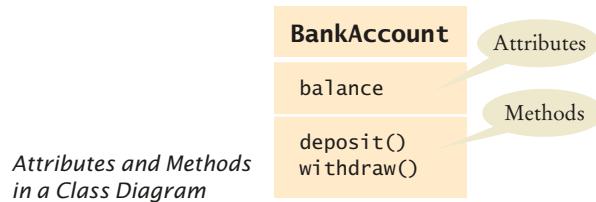
Some dependency relationships give rise to aggregations. For each of the dependency relationships, ask yourself: How does the object locate its collaborator? Does it navigate to it directly because it stores a reference? In that case, draw an aggregation arrow. Or is the collaborator a method parameter variable or return value? Then simply draw a dependency arrow.

Special Topic 12.1**Attributes and Methods in UML Diagrams**

Sometimes it is useful to indicate class *attributes* and *methods* in a class diagram. An **attribute** is an externally observable property that objects of a class have. For example, `name` and `price` would be attributes of the `Product` class. Usually, attributes correspond to instance variables. But they don't have to—a class may have a different way of organizing its data. For example, a `GregorianCalendar` object from the Java library has attributes `day`, `month`, and `year`, and it would be appropriate to draw a UML diagram that shows these attributes. However, the class doesn't actually have instance variables that store these quantities. Instead, it internally represents all dates by counting the milliseconds from January 1, 1970—an implementation detail that a class user certainly doesn't need to know about.

You can indicate attributes and methods in a class diagram by dividing a class rectangle into three compartments, with the class name in the top, attributes in the middle, and methods in the bottom (see the figure below). You need not list *all* attributes and methods in a particular diagram. Just list the ones that are helpful for understanding whatever point you are making with a particular diagram.

Also, don't list as an attribute what you also draw as an aggregation. If you denote by aggregation the fact that a Car has Tire objects, don't add an attribute tires.



Special Topic 12.2



Multiplicities

Some designers like to write *multiplicities* at the end(s) of an aggregation relationship to denote how many objects are aggregated. The notations for the most common multiplicities are:

- any number (zero or more): *
- one or more: 1..*
- zero or one: 0..1
- exactly one: 1

The figure below shows that a customer has one or more bank accounts.



An Aggregation Relationship with Multiplicities

Special Topic 12.3



Aggregation, Association, and Composition

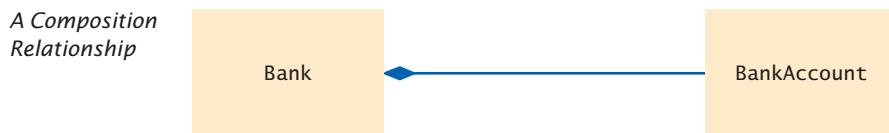
Some designers find the aggregation or *has-a* terminology unsatisfactory. For example, consider customers of a bank. Does the bank “have” customers? Do the customers “have” bank accounts, or does the bank “have” them? Which of these “has” relationships should be modeled by aggregation? This line of thinking can lead us to premature implementation decisions.

Early in the design phase, it makes sense to use a more general relationship between classes called **association**. A class is associated with another if you can *navigate* from objects of one class to objects of the other class. For example, given a Bank object, you can navigate to Customer objects, perhaps by accessing an instance variable, or by making a database lookup.

The UML notation for an association relationship is a solid line, with optional arrows that show in which directions you can navigate the relationship. You can also add words to the line ends to further explain the nature of the relationship. The figure below shows that you can navigate from Bank objects to Customer objects, but you cannot navigate the other way around. That is, in this particular design, the Customer class has no mechanism to determine in which banks it keeps its money.



The UML standard also recognizes a stronger form of the aggregation relationship called **composition**, where the aggregated objects do not have an existence independent of the containing object. For example, composition models the relationship between a bank and its accounts. If a bank closes, the account objects cease to exist as well. In the UML notation, composition looks like aggregation with a filled-in diamond.



Frankly, the differences between aggregation, association, and composition can be confusing, even to experienced designers. If you find the distinction helpful, by all means use the relationship that you find most appropriate. But don't spend time pondering subtle differences between these concepts. From the practical point of view of a Java programmer, it is useful to know when objects of one class have references to objects of another class. The aggregation or *has-a* relationship accurately describes this phenomenon.

Programming Tip 12.1



Make Parallel Arrays into Arrays of Objects

Programmers who are familiar with arrays, but unfamiliar with object-oriented programming, sometimes distribute information across separate arrays. Here is a typical example: A program needs to manage bank data, consisting of account numbers and balances. Don't store the account numbers and balances in separate arrays.

```
int[] accountNumbers; // Don't do this
double[] balances;
```

Arrays such as these are called **parallel arrays** (see Figure 7). The *i*th slice (`accountNumbers[i]` and `balances[i]`) contains data that need to be processed together.

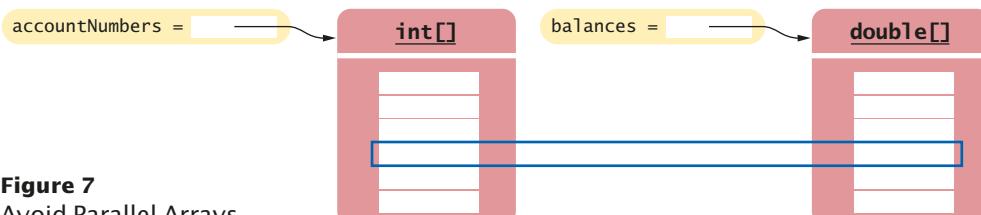


Figure 7
Avoid Parallel Arrays

If you find yourself using two arrays that have the same length, ask yourself whether you couldn't replace them with a single array of a class type. Look at a slice and find the concept that it represents. Then make the concept into a class. In our example each slice contains an account number and a balance, describing a bank account. Therefore, it is an easy matter to use a single array of objects (see Figure 8):

```
BankAccount[] accounts;
```

Avoid parallel arrays
by changing them
into arrays of objects.

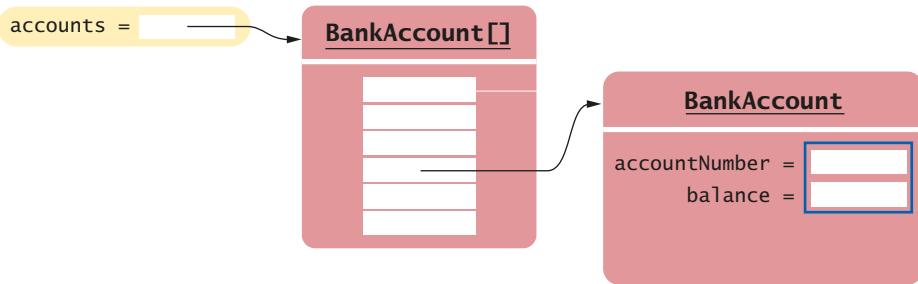


Figure 8 Reorganizing Parallel Arrays into an Array of Objects

Why is this beneficial? Think ahead. Maybe your program will change and you will need to store the owner of the bank account as well. It is a simple matter to update the `BankAccount` class. It may well be quite complicated to add a new array and make sure that all methods that accessed the original two arrays now also correctly access the third one.

12.3 Application: Printing an Invoice

In this book, we discuss a five-part program development process that is particularly well suited for beginning programmers:

1. Gather requirements.
2. Use CRC cards to find classes, responsibilities, and collaborators.
3. Use UML diagrams to record class relationships.
4. Use javadoc to document method behavior.
5. Implement your program.

There isn't a lot of notation to learn. The class diagrams are simple to draw. The deliverables of the design phase are useful for the implementation phase—you simply take the source files and start adding the method code. Of course, as your projects get more complex, you will want to learn more about formal design methods. There are many techniques to describe object scenarios, call sequencing, the large-scale structure of programs, and so on, that are beneficial even for relatively simple projects. *The Unified Modeling Language User Guide* gives a good overview of these techniques.

In this section, we will walk through the object-oriented design technique with a very simple example. In this case, the methodology may feel overblown, but it is a good introduction to the mechanics of each step. You will then be better prepared for the more complex programs that you will encounter in the future.

12.3.1 Requirements

Start the development process by gathering and documenting program requirements.

Before you begin designing a solution, you should gather all requirements for your program in plain English. Write down what your program should do. It is helpful to include typical scenarios in addition to a general description.

The task of our sample program is to print out an invoice. An invoice describes the charges for a set of products in certain quantities. (We omit complexities such as dates, taxes, and invoice and customer numbers.) The program simply prints the



© Scott Cramer/Stockphoto.

An invoice lists the charges for each item and the amount due.

billing address, all line items, and the amount due. Each line item contains the description and unit price of a product, the quantity ordered, and the total price. Also, in the interest of simplicity, we do not provide a user interface. We just supply a test program that adds line items to the invoice and then prints it.

I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description	Price	Qty	Total
Toaster	29.95	3	89.85
Hair dryer	24.95	1	24.95
Car vacuum	19.99	2	39.98

AMOUNT DUE: \$154.78

12.3.2 CRC Cards

Use CRC cards to find classes, responsibilities, and collaborators.

When designing an object-oriented program, you need to discover classes. Classes correspond to nouns in the requirements specification. In this problem, it is pretty obvious what the nouns are:

Invoice	Address	LineItem	Product	Description
Price	Quantity	Total	Amount due	

(Of course, Toaster doesn't count—it is the description of a LineItem object and therefore a data value, not the name of a class.)

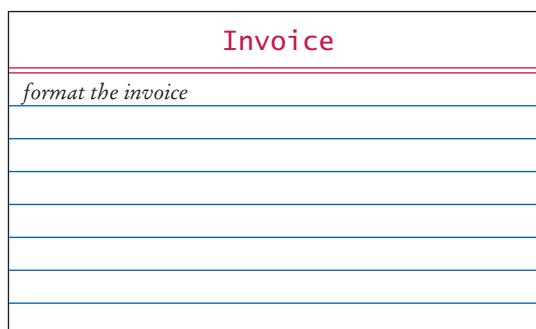
Description and price are attributes of the Product class. What about the quantity? The quantity is not an attribute of a Product. Just as in the printed invoice, let's have a class LineItem that records the product and the quantity (such as "3 toasters").

The total and amount due are computed—not stored anywhere. Thus, they don't lead to classes. After this process of elimination, we are left with four candidates for classes:

Invoice	Address	LineItem	Product
---------	---------	----------	---------

Each of them represents a useful concept, so let's make them all into classes.

The purpose of the program is to print an invoice. However, the Invoice class won't necessarily know whether to display the output in `System.out`, in a text area, or in a file. Therefore, let's relax the task slightly and make the invoice responsible for *formatting* the invoice. The result is a string (containing multiple lines) that can be printed out or displayed. Record that responsibility on a CRC card:

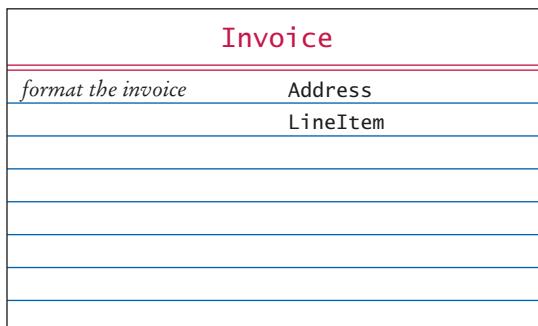


How does an invoice format itself? It must format the billing address, format all line items, and then add the amount due. How can the invoice format an address? It can't—that really is the responsibility of the `Address` class. This leads to a second CRC card:



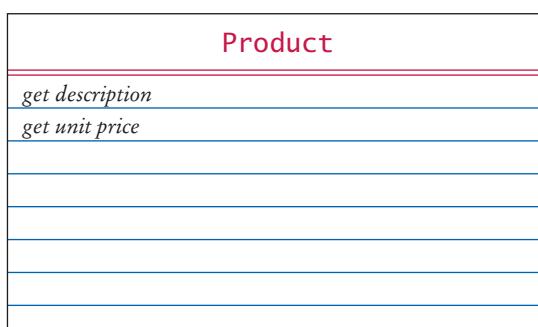
Similarly, formatting of a line item is the responsibility of the `LineItem` class.

The `format` method of the `Invoice` class calls the `format` methods of the `Address` and `LineItem` classes. Whenever a method uses another class, you list that other class as a collaborator. In other words, `Address` and `LineItem` are collaborators of `Invoice`:



When formatting the invoice, the invoice also needs to compute the total amount due. To obtain that amount, it must ask each line item about the total price of the item.

How does a line item obtain that total? It must ask the product for the unit price, and then multiply it by the quantity. That is, the `Product` class must reveal the unit price, and it is a collaborator of the `LineItem` class.



LineItem	
format the item	Product
get total price	

Finally, the invoice must be populated with products and quantities, so that it makes sense to format the result. That too is a responsibility of the `Invoice` class.

Invoice	
format the invoice	Address
add a product and quantity	LineItem
	Product

We now have a set of CRC cards that completes the CRC card process.

12.3.3 UML Diagrams

Use UML diagrams to record class relationships.

After you have discovered classes and their relationships with CRC cards, you should record your findings in a UML diagram. The dependency relationships come from the collaboration column on the CRC cards. Each class depends on the classes with which it collaborates. In our example, the `Invoice` class collaborates with the `Address`, `LineItem`, and `Product` classes. The `LineItem` class collaborates with the `Product` class.

Now ask yourself which of these dependencies are actually aggregations. How does an invoice know about the address, line item, and product objects with which it collaborates? An invoice object must hold references to the address and the line items when it formats the invoice. But an invoice object need not hold a reference to a product object when adding a product. The product is turned into a line item, and then it is the item's responsibility to hold a reference to it.

Therefore, the `Invoice` class aggregates the `Address` and `LineItem` classes. The `LineItem` class aggregates the `Product` class. However, there is no *has-a* relationship between an invoice and a product. An invoice doesn't store products directly—they are stored in the `LineItem` objects.

There is no inheritance in this example.

Figure 9 shows the class relationships that we discovered.

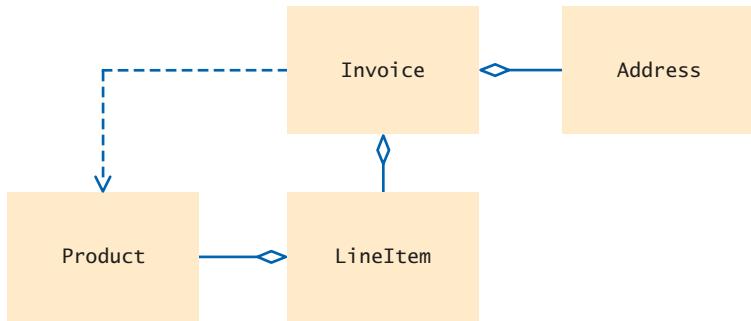


Figure 9 The Relationships Between the Invoice Classes

12.3.4 Method Documentation

Use javadoc comments (with the method bodies left blank) to record the behavior of classes.

The final step of the design phase is to write the documentation of the discovered classes and methods. Simply write a Java source file for each class, write the method comments for those methods that you have discovered, and leave the bodies of the methods blank.

```


/**
 * Describes an invoice for a set of purchased products.
 */
public class Invoice
{
    /**
     * Adds a charge for a product to this invoice.
     * @param aProduct the product that the customer ordered
     * @param quantity the quantity of the product
     */
    public void add(Product aProduct, int quantity)
    {
    }

    /**
     * Formats the invoice.
     * @return the formatted invoice
     */
    public String format()
    {
    }
}

/**
 * Describes a quantity of an article to purchase.
 */
public class LineItem
{
    /**
     * Computes the total cost of this line item.
     * @return the total price
     */
    public double getTotalPrice()
    {
    }
}


```

```

    }

    /**
     * Formats this item.
     * @return a formatted string of this item
    */
    public String format()
    {
    }
}

/**
 * Describes a product with a description and a price.
 */
public class Product
{
    /**
     * Gets the product description.
     * @return the description
    */
    public String getDescription()
    {
    }

    /**
     * Gets the product price.
     * @return the unit price
    */
    public double getPrice()
    {
    }
}

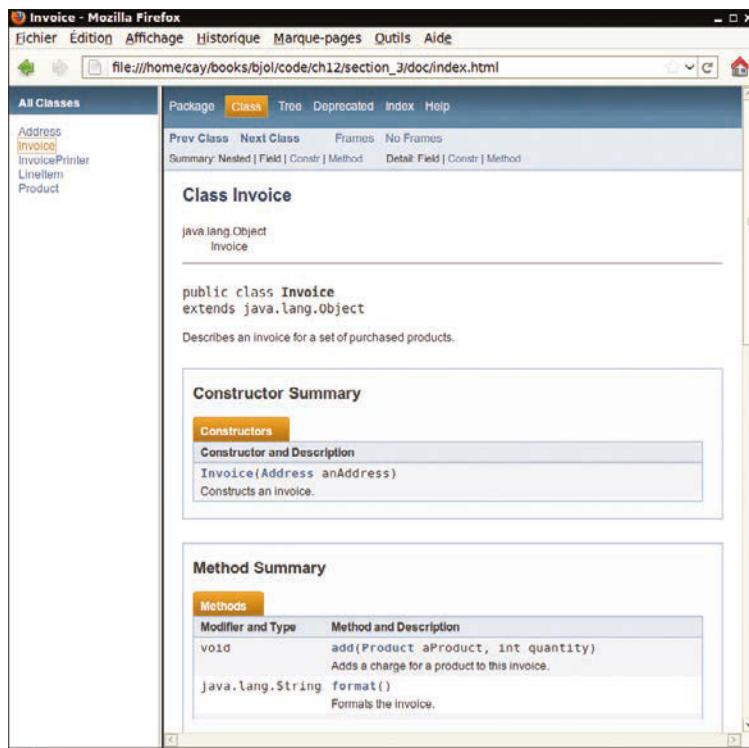
/**
 * Describes a mailing address.
 */
public class Address
{
    /**
     * Formats the address.
     * @return the address as a string with three lines
    */
    public String format()
    {
    }
}

```

Then run the javadoc program to obtain a neatly formatted version of your documentation in HTML format (see Figure 10).

This approach for documenting your classes has a number of advantages. You can share the HTML documentation with others if you work in a team. You use a format that is immediately useful—Java source files that you can carry into the implementation phase. And, most importantly, you supply the comments for the key methods—a task that less prepared programmers leave for later, and often neglect for lack of time.

Figure 10
Class Documentation
in HTML Format



12.3.5 Implementation

After completing the design, implement your classes.

After you have completed the object-oriented design, you are ready to implement the classes.

You already have the method parameter variables and comments from the previous step. Now look at the UML diagram to add instance variables. Aggregated classes yield instance variables. Start with the `Invoice` class. An invoice aggregates `Address` and `LineItem`. Every invoice has one billing address, but it can have many line items. To store multiple `LineItem` objects, you can use an array list. Now you have the instance variables of the `Invoice` class:

```
public class Invoice
{
    private Address billingAddress;
    private ArrayList<LineItem> items;
    ...
}
```

A line item needs to store a `Product` object and the product quantity. That leads to the following instance variables:

```
public class LineItem
{
    private int quantity;
    private Product theProduct;
    ...
}
```

The methods themselves are now easy to implement. Here is a typical example. You already know what the `getTotalPrice` method of the `LineItem` class needs to do—get the unit price of the product and multiply it with the quantity:

```
/*
    Computes the total cost of this line item.
    @return the total price
*/
public double getTotalPrice()
{
    return theProduct.getPrice() * quantity;
}
```

We will not discuss the other methods in detail—they are equally straightforward.

Finally, you need to supply constructors, another routine task.

The entire program is shown below. It is a good practice to go through it in detail and match up the classes and methods to the CRC cards and UML diagram.

Worked Example 12.1 ([at wiley.com/go/bj102examples](http://wiley.com/go/bj102examples)) demonstrates the design process with a more challenging problem, a simulated automatic teller machine. You should download and study that example as well.

In this chapter, you learned a systematic approach for building a relatively complex program. However, object-oriented design is definitely not a spectator sport. To really learn how to design and implement programs, you have to gain experience by repeating this process with your own projects. It is quite possible that you don't immediately home in on a good solution and that you need to go back and reorganize your classes and responsibilities. That is normal and only to be expected. The purpose of the object-oriented design process is to spot these problems in the design phase, when they are still easy to rectify, instead of in the implementation phase, when massive reorganization is more difficult and time consuming.

sec03/InvoicePrinter.java

```
1 /**
2     This program demonstrates the invoice classes by printing
3     a sample invoice.
4 */
5 public class InvoicePrinter
6 {
7     public static void main(String[] args)
8     {
9         Address samsAddress
10        = new Address("Sam's Small Appliances",
11                      "100 Main Street", "Anytown", "CA", "98765");
12
13        Invoice samsInvoice = new Invoice(samsAddress);
14        samsInvoice.add(new Product("Toaster", 29.95), 3);
15        samsInvoice.add(new Product("Hair dryer", 24.95), 1);
16        samsInvoice.add(new Product("Car vacuum", 19.99), 2);
17
18        System.out.println(samsInvoice.format());
19    }
20 }
```

sec03/Invoice.java

```
1 import java.util.ArrayList;
2
```

```
3  /**
4   * Describes an invoice for a set of purchased products.
5  */
6  public class Invoice
7  {
8      private Address billingAddress;
9      private ArrayList<LineItem> items;
10
11     /**
12      Constructs an invoice.
13      @param anAddress the billing address
14     */
15     public Invoice(Address anAddress)
16     {
17         items = new ArrayList<LineItem>();
18         billingAddress = anAddress;
19     }
20
21     /**
22      Adds a charge for a product to this invoice.
23      @param aProduct the product that the customer ordered
24      @param quantity the quantity of the product
25     */
26     public void add(Product aProduct, int quantity)
27     {
28         LineItem anItem = new LineItem(aProduct, quantity);
29         items.add(anItem);
30     }
31
32     /**
33      Formats the invoice.
34      @return the formatted invoice
35     */
36     public String format()
37     {
38         String r = String.format("%32s%n%n", "I N V O I C E")
39                     + billingAddress.format()
40                     + String.format("%n%n%-30s%8s%5s%8s%n",
41                         "Description", "Price", "Qty", "Total");
42
43         for (LineItem item : items)
44         {
45             r = String.format("%s%s%n", r, item.format());
46         }
47
48         r = r + String.format("%nAMOUNT DUE: $%.2f%n", getAmountDue());
49
50         return r;
51     }
52
53     /**
54      Computes the total amount due.
55      @return the amount due
56     */
57     private double getAmountDue()
58     {
59         double amountDue = 0;
60         for (LineItem item : items)
61         {
62             amountDue = amountDue + item.getTotalPrice();
```

```

63     }
64     return amountDue;
65   }
66 }
```

sec03/LineItem.java

```

1  /**
2   * Describes a quantity of an article to purchase.
3  */
4  public class LineItem
5  {
6    private int quantity;
7    private Product theProduct;
8
9    /**
10     * Constructs an item from the product and quantity.
11     * @param aProduct the product
12     * @param aQuantity the item quantity
13    */
14    public LineItem(Product aProduct, int aQuantity)
15    {
16      theProduct = aProduct;
17      quantity = aQuantity;
18    }
19
20   /**
21    * Computes the total cost of this line item.
22    * @return the total price
23   */
24   public double getTotalPrice()
25   {
26     return theProduct.getPrice() * quantity;
27   }
28
29   /**
30    * Formats this item.
31    * @return a formatted string of this line item
32   */
33   public String format()
34   {
35     return String.format("%-30s%8.2f%5d%8.2f",
36       theProduct.getDescription(), theProduct.getPrice(),
37       quantity, getTotalPrice());
38   }
39 }
```

sec03/Product.java

```

1  /**
2   * Describes a product with a description and a price.
3  */
4  public class Product
5  {
6    private String description;
7    private double price;
8
9    /**
10     * Constructs a product from a description and a price.
11     * @param aDescription the product description
12   }
```

```

12     @param aPrice the product price
13 */
14 public Product(String aDescription, double aPrice)
15 {
16     description = aDescription;
17     price = aPrice;
18 }
19
20 /**
21  * Gets the product description.
22  * @return the description
23 */
24 public String getDescription()
25 {
26     return description;
27 }
28
29 /**
30  * Gets the product price.
31  * @return the unit price
32 */
33 public double getPrice()
34 {
35     return price;
36 }
37 }
```

sec03/Address.java

```

1  /**
2   * Describes a mailing address.
3  */
4  public class Address
5  {
6      private String name;
7      private String street;
8      private String city;
9      private String state;
10     private String zip;
11
12 /**
13  * Constructs a mailing address.
14  * @param aName the recipient name
15  * @param aStreet the street
16  * @param aCity the city
17  * @param aState the two-letter state code
18  * @param aZip the ZIP postal code
19 */
20 public Address(String aName, String aStreet,
21                 String aCity, String aState, String aZip)
22 {
23     name = aName;
24     street = aStreet;
25     city = aCity;
26     state = aState;
27     zip = aZip;
28 }
29
30 /**
31  * Formats the address.
32 }
```

```

32     @return the address as a string with three lines
33     */
34     public String format()
35     {
36         return String.format("%s%n%s%n%s", 
37             name, street, city, state, zip);
38     }
39 }
```

SELF CHECK

13. Which class in the invoice printing application is responsible for computing the amount due? What are its collaborators for this task?
14. Why do the format methods return `String` objects instead of directly printing to `System.out`?

Practice It Now you can try these exercises at the end of the chapter: R12.15, E12.4, E12.5.



Computing & Society 12.1 Databases and Privacy

Most companies use computers to keep huge databases of customer records and other business information. Databases not only lower the cost of doing business, they improve the quality of service that companies can offer. Nowadays it is almost unimaginable how time-consuming it used to be to withdraw money from a bank branch or to make travel reservations.

As these databases became ubiquitous, they started creating problems for citizens. Consider the “no fly list” maintained by the U.S. government, which lists names used by suspected terrorists. On March 1, 2007, Professor Walter Murphy, a constitutional scholar of Princeton University and a decorated former Marine, was denied a boarding pass. The airline employee asked him, “Have you been in any peace marches? We ban a lot of people from flying because of that.” As Murphy tells it, “I explained that I had not so marched but had, in September 2006, given a lecture at Princeton, televised and put on the Web, highly critical of George Bush for his many violations of the constitution. ‘That’ll do it,’ the man said.”

We do not actually know if Professor Murphy’s name was on the list because he was critical of the Bush administration or because some other potentially dangerous person had traveled under the same name. Travelers with similar misfortunes had serious

difficulties trying to get themselves off the list.

Problems such as these have become commonplace. Companies and the government routinely merge multiple databases, derive information about us that may be quite inaccurate, and then use that information to make decisions. An insurance company may deny coverage, or charge a higher premium, if it finds that you have too many relatives with a certain disease. You may be denied a job because of a credit or medical report. You do not usually know what information about you is stored or how it is used. In cases where the information can be checked—such as credit reports—it is often difficult to correct errors.

Another issue of concern is privacy. Most people do something, at one time or another in their lives, that they do not want everyone to know about. As judge Louis Brandeis wrote in 1928, “Privacy is the right to be alone—the most comprehensive of rights, and the right most valued by civilized man.” When employers can see your old Facebook posts, divorce lawyers have access to toll road records, and Google mines your e-mails and searches to present you “targeted” advertising, you have little privacy left.

The 1948 “universal declaration of human rights” by the United Nations states, “No one shall be subjected to arbitrary interference with his privacy,

family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.” The United States has surprisingly few legal protections against privacy invasion, apart from federal laws protecting student records and video rentals (the latter was passed after a Supreme Court nominee’s video rental records were published). Other industrialized countries have gone much further and recognize every citizen’s right to control what information about them should be communicated to others and under what circumstances.



© Greg Nicholas/iStockphoto.

If you pay road or bridge tolls with an electronic pass, your records may not be private.



WORKED EXAMPLE 12.1



Simulating an Automatic Teller Machine

Learn to apply the object-oriented design methodology to the simulation of an automatic teller machine that works with both a console-based and graphical user interface. Go to wiley.com/go/bj102examples and download Worked Example 12.1.



© Mark Evans/
iStockphoto.

CHAPTER SUMMARY

Recognize how to discover classes and their responsibilities.



- To discover classes, look for nouns in the problem description.
- Concepts from the problem domain are good candidates for classes.
- A CRC card describes a class, its responsibilities, and its collaborating classes.
- The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

Categorize class relationships and produce UML diagrams that describe them.



- A class depends on another class if it uses objects of that class.
- It is a good practice to minimize the coupling (i.e., dependency) between classes.
- A class aggregates another if its objects contain objects of the other class.
- Inheritance (the *is-a* relationship) is sometimes inappropriately used when the *has-a* relationship would be more appropriate.
- Aggregation (the *has-a* relationship) denotes that objects of one class contain references to objects of another class.
- You need to be able to distinguish the UML notation for inheritance, interface implementation, aggregation, and dependency.

Apply an object-oriented development process to designing a program.

- Start the development process by gathering and documenting program requirements.
- Use CRC cards to find classes, responsibilities, and collaborators.
- Use UML diagrams to record class relationships.
- Use javadoc comments (with the method bodies left blank) to record the behavior of classes.
- After completing the design, implement your classes.

REVIEW EXERCISES

- R12.1** List the steps in the process of object-oriented design that this chapter recommends for student use.
- R12.2** Give a rule of thumb for how to find classes when designing a program.
- R12.3** Give a rule of thumb for how to find methods when designing a program.
- R12.4** After discovering a method, why is it important to identify the object that is *responsible* for carrying out the action?

■■ R12.5 What relationship is appropriate between the following classes: aggregation, inheritance, or neither?

- | | |
|--|---|
| a. University–Student
b. Student–TeachingAssistant
c. Student–Freshman
d. Student–Professor | e. Car–Door
f. Truck–Vehicle
g. Traffic–TrafficSign
h. TrafficSign–Color |
|--|---|

■■ R12.6 Every BMW is a vehicle. Should a class `BMW` inherit from the class `Vehicle`? BMW is a vehicle manufacturer. Does that mean that the class `BMW` should inherit from the class `VehicleManufacturer`?

■■ R12.7 Some books on object-oriented programming recommend using inheritance so that the class `Circle` extends the class `java.awt.Point`. Then the `Circle` class inherits the `setLocation` method from the `Point` superclass. Explain why the `setLocation` method need not be overridden in the subclass. Why is it nevertheless not a good idea to have `Circle` inherit from `Point`? Conversely, would inheriting `Point` from `Circle` fulfill the *is-a* rule? Would it be a good idea?

■ R12.8 Write CRC cards for the `Coin` and `CashRegister` classes described in Section 12.1.3.

■ R12.9 Write CRC cards for the `Quiz` and `Question` classes in Section 12.2.2.

■■ R12.10 Draw a UML diagram for the `Quiz`, `Question`, and `ChoiceQuestion` classes. The `Quiz` class is described in Section 12.2.2.

■■■ R12.11 A file contains a set of records describing countries. Each record consists of the name of the country, its population, and its area. Suppose your task is to write a program that reads in such a file and prints

- The country with the largest area.
- The country with the largest population.
- The country with the largest population density (people per square kilometer).

Think through the problems that you need to solve. What classes and methods will you need? Produce a set of CRC cards, a UML diagram, and a set of javadoc comments.

■■■ R12.12 Discover classes and methods for generating a student report card that lists all classes, grades, and the grade point average for a semester. Produce a set of CRC cards, a UML diagram, and a set of javadoc comments.

■■ R12.13 Consider the following problem description:

Users place coins in a vending machine and select a product by pushing a button. If the inserted coins are sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the inserted coins are returned to the user.

What classes should you use to implement a solution?

■■ R12.14 Consider the following problem description:

Employees receive their biweekly paychecks. They are paid their hourly rates for each hour worked; however, if they worked more than 40 hours per week, they are paid overtime at 150 percent of their regular wage.

What classes should you use to implement a solution?

- R12.15** Consider the following problem description:

Customers order products from a store. Invoices are generated to list the items and quantities ordered, payments received, and amounts still due. Products are shipped to the shipping address of the customer, and invoices are sent to the billing address.

Draw a UML diagram showing the aggregation relationships between the classes `Invoice`, `Address`, `Customer`, and `Product`.

PRACTICE EXERCISES

- E12.1** Enhance the invoice-printing program by providing for two kinds of line items: One kind describes products that are purchased in certain numerical quantities (such as “3 toasters”), another describes a fixed charge (such as “shipping: \$5.00”). *Hint:* Use inheritance. Produce a UML diagram of your modified implementation.
- E12.2** The invoice-printing program is somewhat unrealistic because the formatting of the `LineItem` objects won’t lead to good visual results when the prices and quantities have varying numbers of digits. Enhance the `format` method in two ways: Accept an `int[]` array of column widths as an argument. Use the `NumberFormat` class to format the currency values.
- E12.3** The invoice-printing program has an unfortunate flaw—it mixes “application logic” (the computation of total charges) and “presentation” (the visual appearance of the invoice). To appreciate this flaw, imagine the changes that would be necessary to draw the invoice in HTML for presentation on the Web. Reimplement the program, using a separate `InvoiceFormatter` class to format the invoice. That is, the `Invoice` and `LineItem` methods are no longer responsible for formatting. However, they will acquire other responsibilities, because the `InvoiceFormatter` class needs to query them for the values that it requires.
- E12.4** Write a program that teaches arithmetic to a young child. The program tests addition and subtraction. In level 1, it tests only addition of numbers less than ten whose sum is less than ten. In level 2, it tests addition of arbitrary one-digit numbers. In level 3, it tests subtraction of one-digit numbers with a nonnegative difference.
Generate random problems and get the player’s input. The player gets up to two tries per problem. Advance from one level to the next when the player has achieved a score of five points.
- E12.5** Implement a simple e-mail messaging system. A message has a recipient, a sender, and a message text. A mailbox can store messages. Supply a number of mailboxes for different users and a user interface for users to log in, send messages to other users, read their own messages, and log out. Follow the design process that was described in this chapter.
- E12.6** Modify the implementation of the classes in the ATM simulation in Worked Example 12.1 so that the bank manages a collection of bank accounts and a separate collection of customers. Allow joint accounts in which some accounts can have more than one customer.

PROGRAMMING PROJECTS

- P12.1** Write a program that simulates a vending machine. Products can be purchased by inserting coins with a value at least equal to the cost of the product. A user selects a product from a list of available products, adds coins, and either gets the product or gets the coins returned. The coins are returned if insufficient money was supplied or if the product is sold out. The machine does not give change if too much money was added. Products can be restocked and money removed by an operator. Follow the design process that was described in this chapter. Your solution should include a class `VendingMachine` that is not coupled with the `Scanner` or `PrintStream` classes.
- P12.2** Write a program to design an appointment calendar. An appointment includes the date, starting time, ending time, and a description; for example,

```
Dentist 2017/10/2 17:30 18:30
CS1 class 2017/10/3 08:30 10:00
```

Supply a user interface to add appointments, remove canceled appointments, and print out a list of appointments for a particular day. Follow the design process that was described in this chapter. Your solution should include a class `AppointmentCalendar` that is not coupled with the `Scanner` or `PrintStream` classes.

- P12.3** Write a program that administers and grades quizzes. A quiz consists of questions. There are four types of questions: text questions, number questions, choice questions with a single answer, and choice questions with multiple answers. When grading a text question, ignore leading or trailing spaces and letter case. When grading a numeric question, accept a response that is approximately the same as the answer.
- A quiz is specified in a text file. Each question starts with a letter indicating the question type (T, N, S, M), followed by a line containing the question text. The next line of a non-choice question contains the answer. Choice questions have a list of choices that is terminated by a blank line. Each choice starts with + (correct) or - (incorrect). Here is a sample file:

```
T
Which Java reserved word is used to declare a subclass?
extends
S
What is the original name of the Java language?
- *7
- C--
+ Oak
- Gosling

M
Which of the following types are supertypes of Rectangle?
- PrintStream
+ Shape
+ RectangularShape
+ Object
- String

N
What is the square root of 2?
1.41421356
```

Your program should read in a quiz file, prompt the user for responses to all questions, and grade the responses. Follow the design process described in this chapter.

■■ P12.4 Produce a requirements document for a program that allows a company to send out personalized mailings, either by e-mail or through the postal service. Template files contain the message text, together with variable fields (such as Dear [Title] [Last Name]...). A database (stored as a text file) contains the field values for each recipient. Use HTML as the output file format. Then design and implement the program.

■■■ P12.5 Write a tic-tac-toe game that allows a human player to play against the computer. Your program will play many turns against a human opponent, and it will learn. When it is the computer's turn, the computer randomly selects an empty field, except that it won't ever choose a losing combination. For that purpose, your program must keep an array of losing combinations. Whenever the human wins, the immediately preceding combination is stored as losing. For example, suppose that X = computer and O = human.

Suppose the current combination is

O	X	X
O		

Now it is the human's turn, who will of course choose

O	X	X
O		
		O

The computer should then remember the preceding combination

O	X	X
O		
		O

as a losing combination. As a result, the computer will never again choose that combination from

O	X	
O		
		O

O	X	
O		
		O

Discover classes and supply a UML diagram before you begin to program.

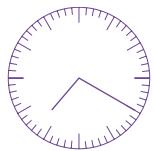
■■■ Business P12.6 *Airline seating.* Write a program that assigns seats on an airplane. Assume the airplane has 20 seats in first class (5 rows of 4 seats each, separated by an aisle) and 90 seats in economy class (15 rows of 6 seats each, separated by an aisle). Your program should take three commands: add passengers, show seating, and quit. When passengers are added, ask for the class (first or economy), the number of passengers traveling together (1 or 2 in first class; 1 to 3 in economy), and the seating preference (aisle or window in first class; aisle, center, or window in economy). Then try to find a match and assign the seats. If no match exists, print a message. Your solution should include a class `Airplane` that is not coupled with the `Scanner` or `PrintStream` classes. Follow the design process that was described in this chapter.

■■■ Business P12.7 In an airplane, each passenger has a touch screen for ordering a drink and a snack. Some items are free and some are not. The system prepares two reports for speeding up service:

1. A list of all seats, ordered by row, showing the charges that must be collected.
2. A list of how many drinks and snacks of each type must be prepared for the front and the rear of the plane.

Follow the design process that was described in this chapter to identify classes, and implement a program that simulates the system.

■■■ Graphics P12.8



An Analog Clock

Implement a program to teach a young child to read the clock. In the game, present an analog clock, such as the one shown at left. Generate random times and display the clock. Accept guesses from the player. Reward the player for correct guesses. After two incorrect guesses, display the correct answer and make a new random time. Implement several levels of play. In level 1, only show full hours. In level 2, show quarter hours. In level 3, show five-minute multiples, and in level 4, show any number of minutes. After a player has achieved five correct guesses at one level, advance to the next level.

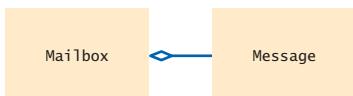
■■■ Graphics P12.9

Write a program that can be used to design a suburban scene, with houses, streets, and cars. Users can add houses and cars of various colors to a street. Write more specific requirements that include a detailed description of the user interface. Then, discover classes and methods, provide UML diagrams, and implement your program.

■■■ Graphics P12.10

Write a simple graphics editor that allows users to add a mixture of shapes (ellipses, rectangles, and lines in different colors) to a panel. Supply commands to load and save the picture. Discover classes, supply a UML diagram, and implement your program.

ANSWERS TO SELF-CHECK QUESTIONS

1. Look for nouns in the problem description.
2. Yes (`ChessBoard`) and no (`MovePiece`).
3. `PrintStream`.
4. To produce the shipping address of the customer.
5. Rework the responsibilities so that they are at a higher level, or come up with more classes to handle the responsibilities.
6. The `CashRegisterTester` class depends on the `CashRegister`, `Coin`, and `System` classes.
7. The `ChoiceQuestion` class inherits from the `Question` class.
8. The `Quiz` class depends on the `Question` class but probably not `ChoiceQuestion`, if we assume that the methods of the `Quiz` class manipulate generic `Question` objects, as they did in Chapter 9.
9. If a class doesn't depend on another, it is not affected by interface changes in the other class.
10. 

```

classDiagram
    class Mailbox
    class Message
    Mailbox "2" --> "1" Message
  
```
11. Typically, a library system wants to track which books a patron has checked out, so it makes more sense to have `Patron` aggregate `Book`. However, there is not always one true answer in design. If you feel strongly that it is important to identify the patron who checked out a particular book (perhaps to notify the patron to return it because it was requested by someone else), then you can argue that the aggregation should go the other way around.
12. There would be no relationship.
13. The `Invoice` class is responsible for computing the amount due. It collaborates with the `LineItem` class.
14. This design decision reduces coupling. It enables us to reuse the classes when we want to show the invoice in a dialog box or on a web page.

WORKED EXAMPLE 12.1

Simulating an Automatic Teller Machine



In this Worked Example, we apply the object-oriented design methodology to the simulation of an automatic teller machine (ATM).

Problem Statement Simulate an ATM that handles checking and savings accounts. Provide both a console-based and graphical user interface.



© Mark Evans/iStockphoto.

Step 1 Gather requirements.

The purpose of this project is to simulate an automatic teller machine. The ATM is used by the customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer also has a customer number and a personal identification number (PIN); both are required to gain access to the accounts. (In a real ATM, the customer number would be recorded on the magnetic strip of the ATM card. In this simulation, the customer will need to type it in.) With the ATM, customers can select an account (checking or savings). The balance of the selected account is displayed. Then the customer can deposit and withdraw money. This process is repeated until the customer chooses to exit.

The details of the user interaction depend on the user interface that we choose for the simulation. We will develop two separate interfaces: a graphical interface that closely mimics an actual ATM (see Figure 11), and a text-based interface that allows you to test the ATM and bank classes without being distracted by GUI programming.

In the GUI interface, the ATM has a keypad to enter numbers, a display to show messages, and a set of buttons, labeled A, B, and C, whose function depends on the state of the machine.

Specifically, the user interaction is as follows. When the ATM starts up, it expects a user to enter a customer number. The display shows the following message:

```
Enter customer number
A = OK
```

The user enters the customer number on the keypad and presses the A button. The display message changes to

```
Enter PIN
A = OK
```

Next, the user enters the PIN and presses the A button again. If the customer number and ID match those of one of the customers in the bank, then the customer can proceed. If not, the user is again prompted to enter the customer number.

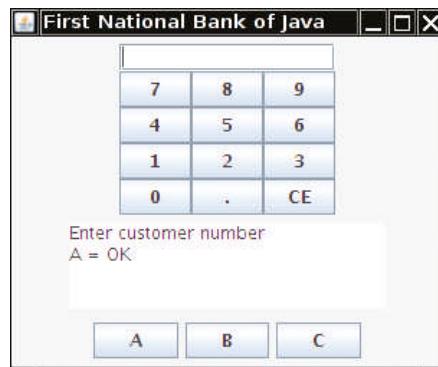


Figure 11
Graphical User Interface for
the Automatic Teller Machine

WE2 Chapter 12 Object-Oriented Design

If the customer has been authorized to use the system, then the display message changes to

Select Account
A = Checking
B = Savings
C = Exit

If the user presses the C button, the ATM reverts to its original state and asks the next user to enter a customer number.

If the user presses the A or B buttons, the ATM remembers the selected account, and the display message changes to

Balance = *balance of selected account*
Enter amount and select transaction
A = Withdraw
B = Deposit
C = Cancel

If the user presses the A or B buttons, the value entered in the keypad is withdrawn from or deposited into the selected account. (This is just a simulation, so no money is dispensed and no deposit is accepted.) Afterward, the ATM reverts to the preceding state, allowing the user to select another account or to exit.

If the user presses the C button, the ATM reverts to the preceding state without executing any transaction.

In the text-based interaction, we read input from `System.in` instead of the buttons. Here is a typical dialog:

```
Enter customer number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
```

In our solution, only the user interface classes are affected by the choice of user interface. The remainder of the classes can be used for both solutions—they are decoupled from the user interface.

Because this is a simulation, the ATM does not actually communicate with a bank. It simply loads a set of customer numbers and PINs from a file. All accounts are initialized with a zero balance.

Step 2 Use CRC cards to find classes, responsibilities, and collaborators.

We will again follow the recipe of Section 12.2 and show how to discover classes, responsibilities, and relationships and how to obtain a detailed design for the ATM program.

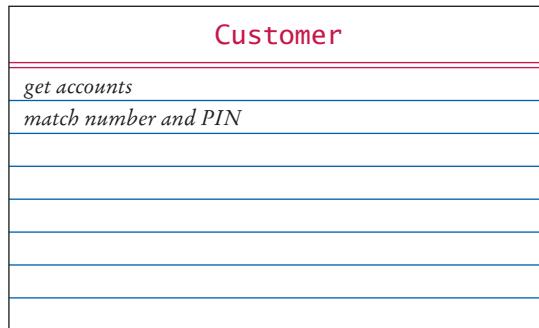
Recall that the first rule for finding classes is “Look for nouns in the problem description”. Here is a list of the nouns:

ATM
User
Keypad
Display
Display message
Button
State
Bank account
Checking account
Savings account
Customer
Customer number
PIN
Bank

Of course, not all of these nouns will become names of classes, and we may yet discover the need for classes that aren't in this list, but it is a good start.

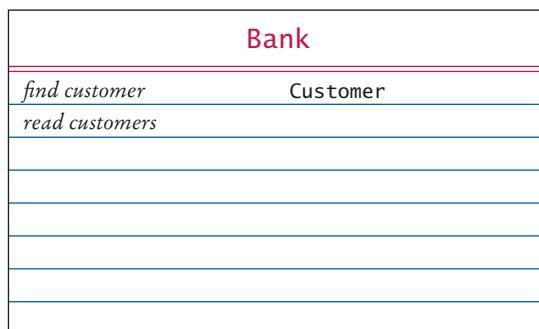
Users and customers represent the same concept in this program. Let's use a class *Customer*. A customer has two bank accounts, and we will require that a *Customer* object should be able to locate these accounts. (Another possible design would make the *Bank* class responsible for locating the accounts of a given customer—see Exercise E12.6.)

A customer also has a customer number and a PIN. We can, of course, require that a customer object give us the customer number and the PIN. But perhaps that isn't so secure. Instead, simply require that a customer object, when given a customer number and a PIN, will tell us whether it matches its own information or not.



A bank contains a collection of customers. When a user walks up to the ATM and enters a customer number and PIN, it is the job of the bank to find the matching customer. How can the bank do this? It needs to check for each customer whether its customer number and PIN match. Thus, it needs to call the *match number and PIN* method of the *Customer* class that we just discovered. Because the *find customer* method calls a *Customer* method, it collaborates with the *Customer* class. We record that fact in the right-hand column of the CRC card.

When the simulation starts up, the bank must also be able to read customer information from a file.

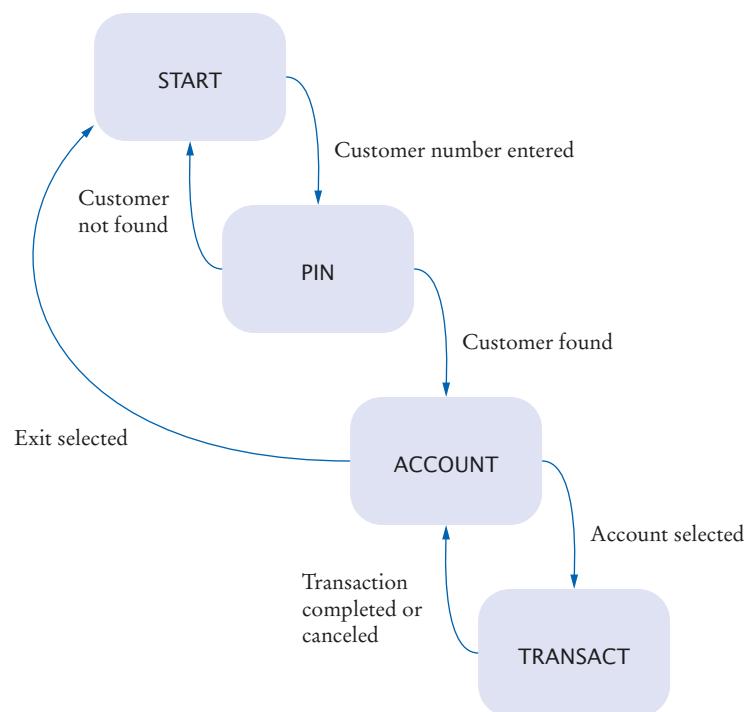


The *BankAccount* class is our familiar class with methods to get the balance and to deposit and withdraw money.

In this program, there is nothing that distinguishes checking accounts from savings accounts. The ATM does not add interest or deduct fees. Therefore, we decide not to implement separate subclasses for checking and savings accounts.

Finally, we are left with the ATM class itself. An important notion of the ATM is the **state**. The current machine state determines the text of the prompts and the function of the buttons. For example, when you first log in, you use the A and B buttons to select an account. Next, you use the same buttons to choose between deposit and withdrawal. The ATM must remember the current state so that it can correctly interpret the buttons.

Figure 12
State Diagram for
the ATM Class



There are four states:

1. **START**: Enter customer ID
2. **PIN**: Enter PIN
3. **ACCOUNT**: Select account
4. **TRANSACT**: Select transaction

To understand how to move from one state to the next, it is useful to draw a **state diagram** (Figure 12). The UML notation has standardized shapes for state diagrams. Draw states as rectangles with rounded corners. Draw state changes as arrows, with labels that indicate the reason for the change.

The user must type a valid customer number and PIN. Then the ATM can ask the bank to find the customer. This calls for a *select customer* method. It collaborates with the bank, asking the bank for the customer that matches the customer number and PIN. Next, there must be a *select account* method that asks the current customer for the checking or savings account. Finally, the ATM must carry out the selected transaction on the current account.

ATM	
manage state	Customer
select customer	Bank
select account	BankAccount
execute transaction	

Of course, discovering these classes and methods was not as neat and orderly as it appears from this discussion. When I designed these classes for this book, it took me several trials and many torn cards to come up with a satisfactory design. It is also important to remember that there is seldom one best design.

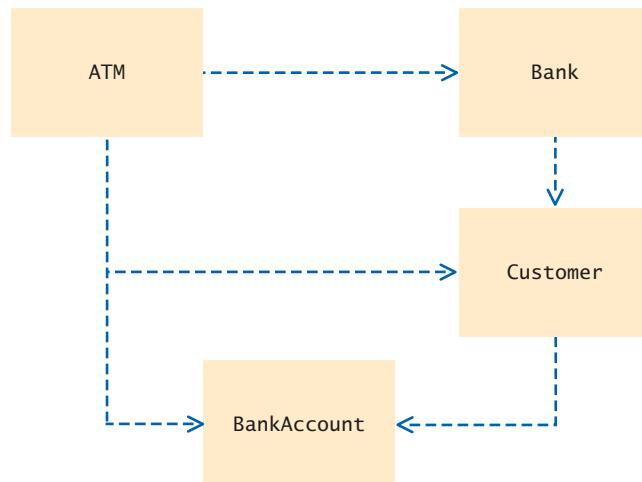
This design has several advantages. The classes describe clear concepts. The methods are sufficient to implement all necessary tasks. (I mentally walked through every ATM usage scenario to verify that.) There are not too many collaboration dependencies between the classes. Thus, I was satisfied with this design and proceeded to the next step.

Step 3

Use UML diagrams to record class relationships.

To draw the dependencies, use the “collaborator” columns from the CRC cards. Looking at those columns, you find that the dependencies are as follows:

- ATM knows about Bank, Customer, and BankAccount.
- Bank knows about Customer.
- Customer knows about BankAccount.



It is easy to see some of the aggregation relationships. A bank has customers, and each customer has two bank accounts.

Does the ATM class aggregate Bank? To answer this question, ask yourself whether an ATM object needs to store a reference to a bank object. Does it need to locate the same bank object across multiple method calls? Indeed it does. Therefore, aggregation is the appropriate relationship.

Does an ATM aggregate customers? Clearly, the ATM is not responsible for storing all of the bank's customers. That's the bank's job. But in our design, the ATM remembers the *current* customer. If a customer has logged in, subsequent commands refer to the same customer. The ATM needs to either store a reference to the customer, or ask the bank to look up the object whenever it needs the current customer. It is a design decision: either store the object, or look it up when needed. We will decide to store the current customer object. That is, we will use aggregation. Note that the choice of aggregation is not an automatic consequence of the problem description—it is a design decision.

Similarly, we will decide to store the current bank account (checking or savings) that the user selects. Therefore, we have an aggregation relationship between ATM and BankAccount.

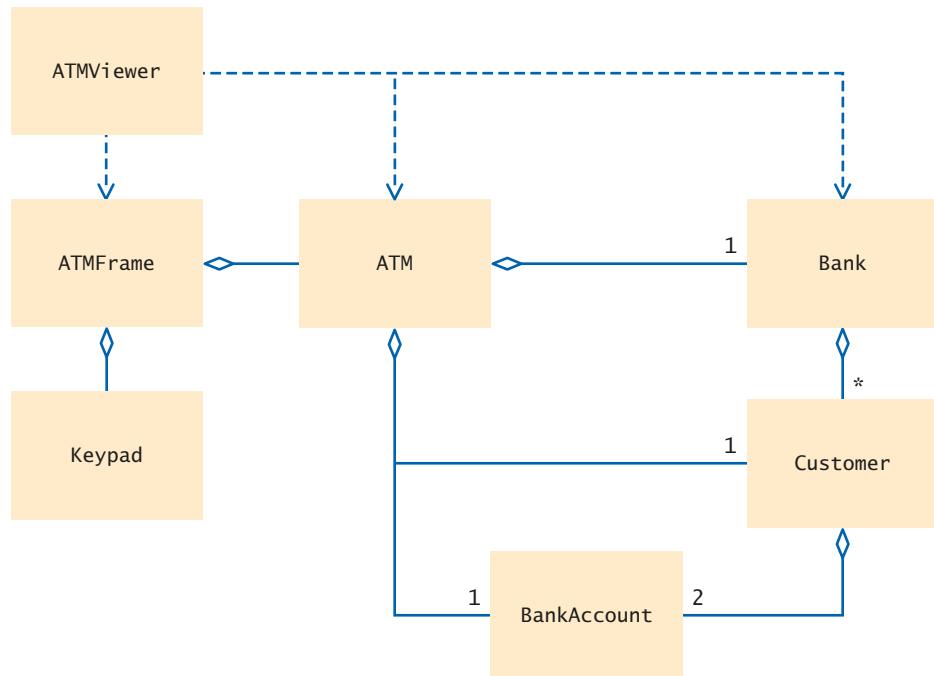


Figure 13 Relationships Between the ATM Classes

Figure 13 shows the relationships between these classes, using the graphical user interface. (The console user interface uses a single class **ATMSimulator** instead of the **ATMFrame**, **ATMViewer**, and **Keypad** classes.)

The class diagram is a good tool to visualize dependencies. Look at the GUI classes. They are completely independent from the rest of the ATM system. You can replace the GUI with a console interface, and you can take out the **Keypad** class and use it in another application. Also, the **Bank**, **BankAccount**, and **Customer** classes, although dependent on each other, don't know anything about the **ATM** class. That makes sense—you can have banks without ATMs. As you can see, when you analyze relationships, you look for both the absence and presence of relationships.

Step 4 Use javadoc to document method behavior.

Now you are ready for the final step of the design phase: documenting the classes and methods that you discovered. Here is a part of the documentation for the **ATM** class:

```

/**
 * An ATM that accesses a bank.
 */
public class ATM
{
    ...
    /**
     * Constructs an ATM for a given bank.
     * @param aBank the bank to which this ATM connects
    */
    public ATM(Bank aBank) { }

    /**
     * Sets the current customer number
     * and sets state to PIN.
     * (Precondition: state is START)
    */
  
```

```

        @param number the customer number
    */
    public void setCustomerNumber(int number) { }

    /**
     * Finds customer in bank.
     * If found sets state to ACCOUNT, else to START.
     * (Precondition: state is PIN)
     * @param pin the PIN of the current customer
    */
    public void selectCustomer(int pin) { }

    /**
     * Sets current account to checking or savings. Sets
     * state to TRANSACT.
     * (Precondition: state is ACCOUNT or TRANSACT)
     * @param account one of CHECKING or SAVINGS
    */
    public void selectAccount(int account) { }

    /**
     * Withdraws amount from current account.
     * (Precondition: state is TRANSACT)
     * @param value the amount to withdraw
    */
    public void withdraw(double value) { }
    ...
}

```

Then run the javadoc utility to turn this documentation into HTML format.

For conciseness, we omit the documentation of the other classes, but they are shown at the end of this example.

Step 5

Implement your program.

Finally, the time has come to implement the ATM simulator. The implementation phase is very straightforward and should take *much less time than the design phase*.

A good strategy for implementing the classes is to go “bottom-up”. Start with the classes that don’t depend on others, such as Keypad and BankAccount. Then implement a class such as Customer that depends only on the BankAccount class. This “bottom-up” approach allows you to test your classes individually. You will find the implementations of these classes at the end of this section.

The most complex class is the ATM class. In order to implement the methods, you need to declare the necessary instance variables. From the class diagram, you can tell that the ATM has a bank object. It becomes an instance variable of the class:

```

public class ATM
{
    private Bank theBank;
    ...
}

```

From the description of the ATM states, it is clear that we require additional instance variables to store the current state, customer, and bank account:

```

public class ATM
{
    private int state;
    private Customer currentCustomer;
    private BankAccount currentAccount;
    ...
}

```

WE8 Chapter 12 Object-Oriented Design

```
}
```

Most methods are very straightforward to implement. Consider the `selectCustomer` method. From the design documentation, we have the description

```
/**  
 * Finds customer in bank.  
 * If found sets state to ACCOUNT, else to START.  
 * (Precondition: state is PIN)  
 * @param pin the PIN of the current customer  
 */
```

This description can be almost literally translated to Java instructions:

```
public void selectCustomer(int pin)  
{  
    currentCustomer = theBank.findCustomer(customerNumber, pin);  
    if (currentCustomer == null)  
    {  
        state = START;  
    }  
    else  
    {  
        state = ACCOUNT;  
    }  
}
```

We won't go through a method-by-method description of the ATM program. You should take some time and compare the actual implementation to the CRC cards and the UML diagram.

worked_example_1/BankAccount.java

```
1  /**  
2   * A bank account has a balance that can be changed by  
3   * deposits and withdrawals.  
4  */  
5  public class BankAccount  
6  {  
7      private double balance;  
8  
9      /**  
10       * Constructs a bank account with a zero balance.  
11      */  
12      public BankAccount()  
13      {  
14          balance = 0;  
15      }  
16  
17      /**  
18       * Constructs a bank account with a given balance.  
19       * @param initialBalance the initial balance  
20      */  
21      public BankAccount(double initialBalance)  
22      {  
23          balance = initialBalance;  
24      }  
25  
26      /**  
27       * Deposits money into the account.  
28       * @param amount the amount of money to withdraw  
29      */
```

```

30    public void deposit(double amount)
31    {
32        balance = balance + amount;
33    }
34
35    /**
36     * Withdraws money from the account.
37     * @param amount the amount of money to deposit
38     */
39    public void withdraw(double amount)
40    {
41        balance = balance - amount;
42    }
43
44    /**
45     * Gets the account balance.
46     * @return the account balance
47     */
48    public double getBalance()
49    {
50        return balance;
51    }
52 }
```

worked_example_1/Customer.java

```

1    /**
2     * A bank customer with a checking and a savings account.
3     */
4    public class Customer
5    {
6        private int customerNumber;
7        private int pin;
8        private BankAccount checkingAccount;
9        private BankAccount savingsAccount;
10
11    /**
12     * Constructs a customer with a given number and PIN.
13     * @param aNumber the customer number
14     * @param aPin the personal identification number
15     */
16    public Customer(int aNumber, int aPin)
17    {
18        customerNumber = aNumber;
19        pin = aPin;
20        checkingAccount = new BankAccount();
21        savingsAccount = new BankAccount();
22    }
23
24    /**
25     * Tests if this customer matches a customer number
26     * and PIN.
27     * @param aNumber a customer number
28     * @param aPin a personal identification number
29     * @return true if the customer number and PIN match
30     */
31    public boolean match(int aNumber, int aPin)
32    {
33        return customerNumber == aNumber && pin == aPin;
```

```

34     }
35
36     /**
37      Gets the checking account of this customer.
38      @return the checking account
39     */
40     public BankAccount getCheckingAccount()
41     {
42         return checkingAccount;
43     }
44
45     /**
46      Gets the savings account of this customer.
47      @return the checking account
48     */
49     public BankAccount getSavingsAccount()
50     {
51         return savingsAccount;
52     }
53 }
```

worked_example_1/Bank.java

```

1  import java.io.File;
2  import java.io.IOException;
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
6  /**
7   * A bank contains customers.
8  */
9  public class Bank
10 {
11     private ArrayList<Customer> customers;
12
13     /**
14      Constructs a bank with no customers.
15     */
16     public Bank()
17     {
18         customers = new ArrayList<Customer>();
19     }
20
21     /**
22      Reads the customer numbers and pins.
23      @param filename the name of the customer file
24     */
25     public void readCustomers(String filename)
26             throws IOException
27     {
28         Scanner in = new Scanner(new File(filename));
29         while (in.hasNext())
30         {
31             int number = in.nextInt();
32             int pin = in.nextInt();
33             Customer c = new Customer(number, pin);
34             addCustomer(c);
35         }
36         in.close();
37     }
```

```

38
39      /**
40       * Adds a customer to the bank.
41       * @param c the customer to add
42     */
43     public void addCustomer(Customer c)
44     {
45         customers.add(c);
46     }
47
48     /**
49      * Finds a customer in the bank.
50      * @param aNumber a customer number
51      * @param aPin a personal identification number
52      * @return the matching customer, or null if no customer
53      * matches
54    */
55    public Customer findCustomer(int aNumber, int aPin)
56    {
57        for (Customer c : customers)
58        {
59            if (c.match(aNumber, aPin))
60            {
61                return c;
62            }
63        }
64        return null;
65    }
66 }
```

worked_example_1/ATM.java

```

1  /**
2   * An ATM that accesses a bank.
3  */
4  public class ATM
5  {
6      public static final int CHECKING = 1;
7      public static final int SAVINGS = 2;
8
9      private int state;
10     private int customerNumber;
11     private Customer currentCustomer;
12     private BankAccount currentAccount;
13     private Bank theBank;
14
15    public static final int START = 1;
16    public static final int PIN = 2;
17    public static final int ACCOUNT = 3;
18    public static final int TRANSACT = 4;
19
20    /**
21     * Constructs an ATM for a given bank.
22     * @param aBank the bank to which this ATM connects
23   */
24   public ATM(Bank aBank)
25   {
26       theBank = aBank;
27       reset();
28   }
```

```
29      /**
30       * Resets the ATM to the initial state.
31       */
32     public void reset()
33     {
34         customerNumber = -1;
35         currentAccount = null;
36         state = START;
37     }
38
39
40     /**
41      * Sets the current customer number
42      * and sets state to PIN.
43      * (Precondition: state is START)
44      * @param number the customer number
45      */
46    public void setCustomerNumber(int number)
47    {
48        customerNumber = number;
49        state = PIN;
50    }
51
52     /**
53      * Finds customer in bank.
54      * If found sets state to ACCOUNT, else to START.
55      * (Precondition: state is PIN)
56      * @param pin the PIN of the current customer
57      */
58    public void selectCustomer(int pin)
59    {
60        currentCustomer
61            = theBank.findCustomer(customerNumber, pin);
62        if (currentCustomer == null)
63        {
64            state = START;
65        }
66        else
67        {
68            state = ACCOUNT;
69        }
70    }
71
72     /**
73      * Sets current account to checking or savings. Sets
74      * state to TRANSACT.
75      * (Precondition: state is ACCOUNT or TRANSACT)
76      * @param account one of CHECKING or SAVINGS
77      */
78    public void selectAccount(int account)
79    {
80        if (account == CHECKING)
81        {
82            currentAccount = currentCustomer.getCheckingAccount();
83        }
84        else
85        {
86            currentAccount = currentCustomer.getSavingsAccount();
87        }
88        state = TRANSACT;
89    }
90 }
```

```

89     }
90
91     /**
92      Withdraws amount from current account.
93      (Precondition: state is TRANSACT)
94      @param value the amount to withdraw
95     */
96     public void withdraw(double value)
97     {
98         currentAccount.withdraw(value);
99     }
100
101    /**
102     Deposits amount to current account.
103     (Precondition: state is TRANSACT)
104     @param value the amount to deposit
105    */
106    public void deposit(double value)
107    {
108        currentAccount.deposit(value);
109    }
110
111    /**
112     Gets the balance of the current account.
113     (Precondition: state is TRANSACT)
114     @return the balance
115    */
116    public double getBalance()
117    {
118        return currentAccount.getBalance();
119    }
120
121    /**
122     Moves back to the previous state.
123    */
124    public void back()
125    {
126        if (state == TRANSACT)
127        {
128            state = ACCOUNT;
129        }
130        else if (state == ACCOUNT)
131        {
132            state = PIN;
133        }
134        else if (state == PIN)
135        {
136            state = START;
137        }
138    }
139
140    /**
141     Gets the current state of this ATM.
142     @return the current state
143    */
144    public int getState()
145    {
146        return state;
147    }
148 }

```

The following class implements a console-based user interface for the ATM.

worked_example_1/ATMSimulator.java

```

1 import java.io.IOException;
2 import java.util.Scanner;
3
4 /**
5  * A text-based simulation of an automatic teller machine.
6 */
7 public class ATMSimulator
8 {
9     public static void main(String[] args)
10    {
11        ATM theATM;
12        try
13        {
14            Bank theBank = new Bank();
15            theBank.readCustomers("customers.txt");
16            theATM = new ATM(theBank);
17        }
18        catch (IOException e)
19        {
20            System.out.println("Error opening accounts file.");
21            return;
22        }
23
24        Scanner in = new Scanner(System.in);
25
26        while (true)
27        {
28            int state = theATM.getState();
29            if (state == ATM.START)
30            {
31                System.out.print("Enter customer number: ");
32                int number = in.nextInt();
33                theATM.setCustomerNumber(number);
34            }
35            else if (state == ATM.PIN)
36            {
37                System.out.print("Enter PIN: ");
38                int pin = in.nextInt();
39                theATM.selectCustomer(pin);
40            }
41            else if (state == ATM.ACCOUNT)
42            {
43                System.out.print("A=Checking, B=Savings, C=Quit: ");
44                String command = in.next();
45                if (command.equalsIgnoreCase("A"))
46                {
47                    theATM.selectAccount(ATM.CHECKING);
48                }
49                else if (command.equalsIgnoreCase("B"))
50                {
51                    theATM.selectAccount(ATM.SAVINGS);
52                }
53                else if (command.equalsIgnoreCase("C"))
54                {
55                    theATM.reset();
56                }
57            }
58        }
59    }
60}
```

```

56     }
57     else
58     {
59         System.out.println("Illegal input!");
60     }
61 }
62 else if (state == ATM.TRANSACT)
63 {
64     System.out.println("Balance=" + theATM.getBalance());
65     System.out.print("A=Deposit, B=Withdrawal, C=Cancel: ");
66     String command = in.nextLine();
67     if (command.equalsIgnoreCase("A"))
68     {
69         System.out.print("Amount: ");
70         double amount = in.nextDouble();
71         theATM.deposit(amount);
72         theATM.back();
73     }
74     else if (command.equalsIgnoreCase("B"))
75     {
76         System.out.print("Amount: ");
77         double amount = in.nextDouble();
78         theATM.withdraw(amount);
79         theATM.back();
80     }
81     else if (command.equalsIgnoreCase("C"))
82     {
83         theATM.back();
84     }
85     else
86     {
87         System.out.println("Illegal input!");
88     }
89 }
90 }
91 }
92 }

```

Program Run

```

Enter customer number: 1
Enter PIN: 1234
A=Checking, B=Savings, C=Quit: A
Balance=0.0
A=Deposit, B=Withdrawal, C=Cancel: A
Amount: 1000
A=Checking, B=Savings, C=Quit: C
...

```

Here are the user-interface classes for the GUI version of the user interface.

worked_example_1/KeyPad.java

```

1 import java.awt.BorderLayout;
2 import java.awt.GridLayout;
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5 import javax.swing.JButton;
6 import javax.swing.JPanel;
7 import javax.swing.JTextField;

```

```
8  /**
9   * A component that lets the user enter a number, using
10  * a button pad labeled with digits.
11  */
12 public class KeyPad extends JPanel
13 {
14     private JPanel buttonPanel;
15     private JButton clearButton;
16     private JTextField display;
17
18     /**
19      * Constructs the keypad panel.
20     */
21     public KeyPad()
22     {
23         setLayout(new BorderLayout());
24
25         // Add display field
26
27         display = new JTextField();
28         add(display, "North");
29
30         // Make button panel
31
32         buttonPanel = new JPanel();
33         buttonPanel.setLayout(new GridLayout(4, 3));
34
35         // Add digit buttons
36
37         addButton("7");
38         addButton("8");
39         addButton("9");
40         addButton("4");
41         addButton("5");
42         addButton("6");
43         addButton("1");
44         addButton("2");
45         addButton("3");
46         addButton("0");
47         addButton(".");
48
49         // Add clear entry button
50
51         clearButton = new JButton("CE");
52         buttonPanel.add(clearButton);
53
54         class ClearButtonListener implements ActionListener
55         {
56             public void actionPerformed(ActionEvent event)
57             {
58                 display.setText("");
59             }
60         }
61         ActionListener listener = new ClearButtonListener();
62
63         clearButton.addActionListener(new
64             ClearButtonListener());
65
66         add(buttonPanel, "Center");
67 }
```

```

68 }
69
70 /**
71  * Adds a button to the button panel.
72  * @param label the button label
73 */
74 private void addButton(final String label)
75 {
76     class DigitButtonListener implements ActionListener
77     {
78         public void actionPerformed(ActionEvent event)
79         {
80             // Don't add two decimal points
81             if (label.equals("."))
82                 && display.getText().indexOf(".") != -1)
83             {
84                 return;
85             }
86
87             // Append label text to button
88             display.setText(display.getText() + label);
89         }
90     }
91
92     JButton button = new JButton(label);
93     buttonPanel.add(button);
94     ActionListener listener = new DigitButtonListener();
95     button.addActionListener(listener);
96 }
97
98 /**
99  * Gets the value that the user entered.
100 * @return the value in the text field of the keypad
101 */
102 public double getValue()
103 {
104     return Double.parseDouble(display.getText());
105 }
106
107 /**
108  * Clears the display.
109 */
110 public void clear()
111 {
112     display.setText("");
113 }
114 }
```

worked_example_1/ATMFrame.java

```

1 import java.awt.FlowLayout;
2 import java.awt.GridLayout;
3 import java.awt.event.ActionEvent;
4 import java.awt.event.ActionListener;
5 import javax.swing.JButton;
6 import javax.swing.JFrame;
7 import javax.swing.JPanel;
8 import javax.swing.JTextArea;
9
```

```
10  /**
11   * A frame displaying the components of an ATM.
12  */
13 public class ATMFrame extends JFrame
14 {
15     private static final int FRAME_WIDTH = 300;
16     private static final int FRAME_HEIGHT = 300;
17
18     private JButton aButton;
19     private JButton bButton;
20     private JButton cButton;
21
22     private KeyPad pad;
23     private JTextArea display;
24
25     private ATM theATM;
26
27     /**
28      * Constructs the user interface of the ATM frame.
29     */
30     public ATMFrame(ATM anATM)
31     {
32         theATM = anATM;
33
34         // Construct components
35         pad = new KeyPad();
36
37         display = new JTextArea(4, 20);
38
39         aButton = new JButton(" A ");
40         aButton.addActionListener(new AButtonListener());
41
42         bButton = new JButton(" B ");
43         bButton.addActionListener(new BButtonListener());
44
45         cButton = new JButton(" C ");
46         cButton.addActionListener(new CButtonListener());
47
48         // Add components
49
50         JPanel buttonPanel = new JPanel();
51         buttonPanel.add(aButton);
52         buttonPanel.add(bButton);
53         buttonPanel.add(cButton);
54
55         setLayout(new FlowLayout());
56         add(pad);
57         add(display);
58         add(buttonPanel);
59         showState();
60
61         setSize(FRAME_WIDTH, FRAME_HEIGHT);
62     }
63
64     /**
65      * Updates display message.
66     */
67     public void showState()
68     {
69         int state = theATM.getState();
```

```

70     pad.clear();
71     if (state == ATM.START)
72     {
73         display.setText("Enter customer number\nA = OK");
74     }
75     else if (state == ATM.PIN)
76     {
77         display.setText("Enter PIN\nA = OK");
78     }
79     else if (state == ATM.ACCOUNT)
80     {
81         display.setText("Select Account\n"
82                         + "A = Checking\nB = Savings\nC = Exit");
83     }
84     else if (state == ATM.TRANSACT)
85     {
86         display.setText("Balance = "
87                         + theATM.getBalance()
88                         + "\nEnter amount and select transaction\n"
89                         + "A = Withdraw\nB = Deposit\nC = Cancel");
90     }
91 }
92
93 class AButtonListener implements ActionListener
94 {
95     public void actionPerformed(ActionEvent event)
96     {
97         int state = theATM.getState();
98         if (state == ATM.START)
99         {
100            theATM.setCustomerNumber((int) pad.getValue());
101        }
102        else if (state == ATM.PIN)
103        {
104            theATM.selectCustomer((int) pad.getValue());
105        }
106        else if (state == ATM.ACCOUNT)
107        {
108            theATM.selectAccount(ATM.CHECKING);
109        }
110        else if (state == ATM.TRANSACT)
111        {
112            theATM.withdraw(pad.getValue());
113            theATM.back();
114        }
115        showState();
116    }
117 }
118
119 class BButtonListener implements ActionListener
120 {
121     public void actionPerformed(ActionEvent event)
122     {
123         int state = theATM.getState();
124         if (state == ATM.ACCOUNT)
125         {
126             theATM.selectAccount(ATM.SAVINGS);
127         }
128         else if (state == ATM.TRANSACT)
129         {

```

```

130         theATM.deposit(pad.getValue());
131         theATM.back();
132     }
133     showState();
134 }
135 }
136
137 class CButtonListener implements ActionListener
138 {
139     public void actionPerformed(ActionEvent event)
140     {
141         int state = theATM.getState();
142         if (state == ATM.ACCOUNT)
143         {
144             theATM.reset();
145         }
146         else if (state == ATM.TRANSACT)
147         {
148             theATM.back();
149         }
150         showState();
151     }
152 }
153 }
```

worked_example_1/ATMViewer.java

```

1 import java.io.IOException;
2 import javax.swing.JFrame;
3 import javax.swing.JOptionPane;
4
5 /**
6  * A graphical simulation of an automatic teller machine.
7 */
8 public class ATMViewer
9 {
10     public static void main(String[] args)
11     {
12         ATM theATM;
13
14         try
15         {
16             Bank theBank = new Bank();
17             theBank.readCustomers("customers.txt");
18             theATM = new ATM(theBank);
19         }
20         catch (IOException e)
21         {
22             JOptionPane.showMessageDialog(null, "Error opening accounts file.");
23             return;
24         }
25
26         JFrame frame = new ATMFrame(theATM);
27         frame.setTitle("First National Bank of Java");
28         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29         frame.setVisible(true);
30     }
31 }
```

RECURSION

CHAPTER GOALS

- To learn to “think recursively”
- To be able to use recursive helper methods
- To understand the relationship between recursion and iteration
- To understand when the use of recursion affects the efficiency of an algorithm
- To analyze problems that are much easier to solve by recursion than by iteration
- To process data with recursive structures using mutual recursion

© Nicolae Popovici/iStockphoto.



CHAPTER CONTENTS

13.1 TRIANGLE NUMBERS 608

CE1 Infinite Recursion 612

CE2 Tracing Through Recursive Methods 612

HT1 Thinking Recursively 613

WE1 Finding Files

13.2 RECURSIVE HELPER METHODS 616

13.3 THE EFFICIENCY OF RECURSION 618

13.4 PERMUTATIONS 623

C&S The Limits of Computation 626

13.5 MUTUAL RECURSION 628

13.6 BACKTRACKING 634

WE2 Towers of Hanoi



© Nicolae Popovici/iStockphoto.

The method of recursion is a powerful technique for breaking up complex computational problems into simpler, often smaller, ones. The term “recursion” refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you both simple and complex examples of recursion and teaches you how to “think recursively”.

13.1 Triangle Numbers

Chapter 5 contains a simple introduction to writing recursive methods—methods that call themselves with simpler inputs. In that chapter, you saw how to print triangle patterns such as this one:

```
[]
[] []
[] [] []
```

In this section, we will modify the example slightly and use recursion to compute the area of a triangle of side length n , assuming that each $[]$ square has area 1. This value is sometimes called the *n th triangle number*. For example, as you can tell from looking at the triangle above, the third triangle number is 6.

We will develop an object-oriented solution that gives another perspective on recursive problem solving. Instead of calling a method with simpler values, we will construct a simpler object.

Here is the outline of the class that we will develop:

```
public class Triangle
{
    private int width;

    public Triangle(int aWidth)
    {
        width = aWidth;
    }

    public int getArea()
    {
        ...
    }
}
```

If the width of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let’s take care of this case first:

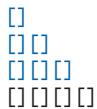
```
public int getArea()
{
    if (width == 1) { return 1; }
    ...
}
```



© Davis Mantei/iStockphoto.

Using the same method as the one in this section, you can compute the volume of a Mayan pyramid.

To deal with the general case, consider this picture:



Suppose we knew the area of the smaller, colored triangle. Then we could easily compute the area of the larger triangle as

$$\text{smallerArea} + \text{width}$$

How can we get the smaller area? Let's make a smaller triangle and ask it!

```
Triangle smallerTriangle = new Triangle(width - 1);
int smallerArea = smallerTriangle.getArea();
```

Now we can complete the `getArea` method:

```
public int getArea()
{
    if (width == 1) { return 1; }
    else
    {
        Triangle smallerTriangle = new Triangle(width - 1);
        int smallerArea = smallerTriangle.getArea();
        return smallerArea + width;
    }
}
```

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

Here is an illustration of what happens when we compute the area of a triangle of width 4:

- The `getArea` method makes a smaller triangle of width 3.
- It calls `getArea` on that triangle.
 - That method makes a smaller triangle of width 2.
 - It calls `getArea` on that triangle.
 - That method makes a smaller triangle of width 1.
 - It calls `getArea` on that triangle.
 - That method returns 1.
 - The method returns `smallerArea + width = 1 + 2 = 3`.
 - The method returns `smallerArea + width = 3 + 3 = 6`.
 - The method returns `smallerArea + width = 6 + 4 = 10`.

This solution has one remarkable aspect. To solve the area problem for a triangle of a given width, we use the fact that we can solve the same problem for a lesser width. This is called a *recursive* solution.

The call pattern of a **recursive method** looks complicated, and the key to the successful design of a recursive method is *not to think about it*. Instead, look at the `getArea` method one more time and notice how utterly reasonable it is. If the width is 1, then, of course, the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle *and don't think about why that works*. Then the area of the larger triangle is clearly the sum of the smaller area and the width.

There are two key requirements to make sure that the recursion is successful:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

For a recursion to terminate, there must be special cases for the simplest values.

The `getArea` method calls itself again with smaller and smaller width values. Eventually the width must reach 1, and there is a special case for computing the area of a triangle with width 1. Thus, the `getArea` method always succeeds.

Actually, you have to be careful. What happens when you call the area of a triangle with width -1 ? It computes the area of a triangle with width -2 , which computes the area of a triangle with width -3 , and so on. To avoid this, the `getArea` method should return 0 if the width ≤ 0 .

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

$$1 + 2 + 3 + \dots + \text{width}$$

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= width; i++)
{
    area = area + i;
}
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first n integers can be computed as

$$1 + 2 + \dots + n = n \times (n + 1)/2$$

Thus, the area equals

$$\text{width} * (\text{width} + 1) / 2$$

Therefore, neither recursion nor a loop is required to solve this problem. The recursive solution is intended as a “warm-up” to introduce you to the concept of recursion.

sec01/Triangle.java

```
1  /**
2   * A triangular shape composed of stacked unit squares like this:
3   * []
4   * []
5   * []
6   * ...
7  */
8  public class Triangle
9  {
10     private int width;
11
12    /**
13     * Constructs a triangular shape.
14     * @param aWidth the width (and height) of the triangle
15     */
16    public Triangle(int aWidth)
17    {
18        width = aWidth;
19    }
20
21    /**
22     * Computes the area of the triangle.
23     * @return the area
24     */
25    public int getArea()
26    {
```

```

27     if (width <= 0) { return 0; }
28     else if (width == 1) { return 1; }
29     else
30     {
31         Triangle smallerTriangle = new Triangle(width - 1);
32         int smallerArea = smallerTriangle.getArea();
33         return smallerArea + width;
34     }
35 }
36 }
```

sec01/TriangleTester.java

```

1  public class TriangleTester
2  {
3      public static void main(String[] args)
4      {
5          Triangle t = new Triangle(10);
6          int area = t.getArea();
7          System.out.println("Area: " + area);
8          System.out.println("Expected: 55");
9      }
10 }
```

Program Run

Area: 55
Expected: 55

SELF CHECK

- Why is the statement `else if (width == 1) { return 1; }` in the final version of the `getArea` method unnecessary?
- How would you modify the program to recursively compute the area of a square?
- In some cultures, numbers containing the digit 8 are lucky numbers. What is wrong with the following method that tries to test whether a number is lucky?

```
public static boolean isLucky(int number)
{
    int lastDigit = number % 10;
    if (lastDigit == 8) { return true; }
    else
    {
        return isLucky(number / 10); // Test the number without the last digit
    }
}
```

- In order to compute a power of two, you can take the next-lower power and double it. For example, if you want to compute 2^{11} and you know that $2^{10} = 1024$, then $2^{11} = 2 \times 1024 = 2048$. Write a recursive method `public static int pow2(int n)` that is based on this observation.
- Consider the following recursive method:

```
public static int mystery(int n)
{
    if (n <= 0) { return 0; }
    else
    {
```

```

        int smaller = n - 1;
        return mystery(smaller) + n * n;
    }
}

```

What is `mystery(4)`?

Practice It Now you can try these exercises at the end of the chapter: E13.1, E13.2, E13.12.

Common Error 13.1



Infinite Recursion

A common programming error is an *infinite recursion*: a method calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is available for this purpose is exhausted. Your program shuts down and reports a “stack overflow”.

Infinite recursion happens either because the arguments don’t get simpler or because a special terminating case is missing. For example, suppose the `getArea` method was allowed to compute the area of a triangle with width 0. If it weren’t for the special test, the method would construct triangles with width -1, -2, -3, and so on.

Common Error 13.2



Tracing Through Recursive Methods

Debugging a recursive method can be somewhat challenging. When you set a **breakpoint** in a recursive method, the program stops as soon as that program line is encountered in *any call to the recursive method*. Suppose you want to debug the recursive `getArea` method of the `Triangle` class. Debug the `TriangleTester` program and run until the beginning of the `getArea` method. Inspect the `width` instance variable. It is 10.

Remove the breakpoint and now run until the statement `return smallerArea + width;` (see Figure 1). When you inspect `width` again, its value is 2! That makes no sense. There was no instruction that changed the value of `width`. Is that a bug with the debugger?

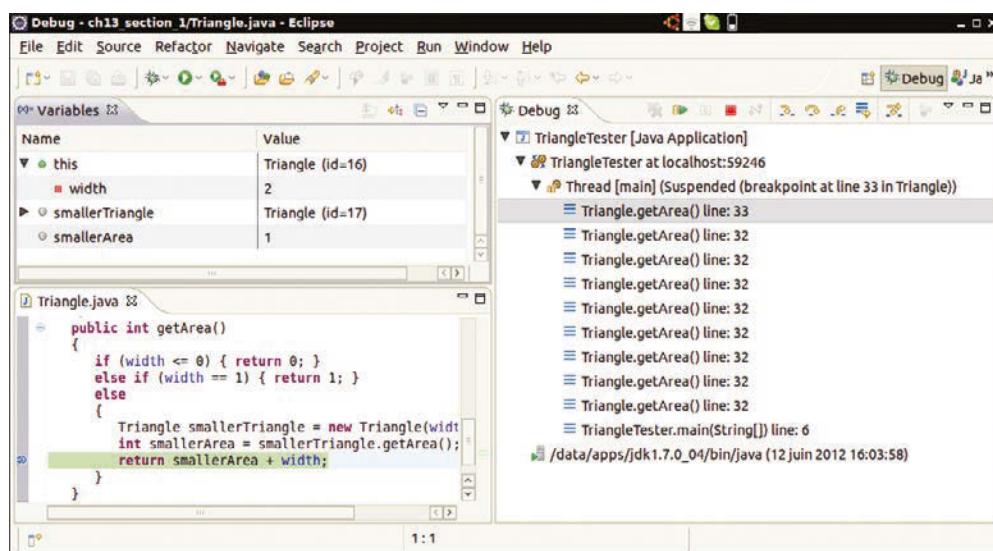


Figure 1 Debugging a Recursive Method

No. The program stopped in the first recursive call to `getArea` that reached the `return` statement. If you are confused, look at the **call stack** (top right in the figure). You will see that nine calls to `getArea` are pending.

You can debug recursive methods with the debugger. You just need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

HOW TO 13.1



Thinking Recursively

Solving a problem recursively requires a different mindset than solving it by programming a loop. In fact, it helps if you pretend to be a bit lazy, asking others to do most of the work for you. If you need to solve a complex problem, pretend that “someone else” will do most of the heavy lifting and solve the problem for simpler inputs. Then you only need to figure out how you can turn the solutions with simpler inputs into a solution for the whole problem. To illustrate the technique of recursion, let us consider the following problem.

Problem Statement Test whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters.



© Nikada/iStockphoto.

Thinking recursively is easy if you can recognize a subtask that is similar to the original task.

Typical examples of palindromes are

- A man, a plan, a canal—Panama!
 - Go hang a salami, I’m a lasagna hog
- and, of course, the oldest palindrome of all:
- Madam, I’m Adam

When testing for a palindrome, we match upper- and lowercase letters, and ignore all spaces and punctuation marks.

We want to implement the `isPalindrome` method in the following class:

```
public class Palindromes
{
    ...
    /**
     * Tests whether a text is a palindrome.
     * @param text a string that is being checked
     * @return true if text is a palindrome, false otherwise
     */
    public static boolean isPalindrome(String text)
    {
        ...
    }
}
```

Step 1 Break the input into parts that can themselves be inputs to the problem.

In your mind, focus on a particular input or set of inputs for the problem that you want to solve. Think how you can simplify the inputs in such a way that the same problem can be applied to the simpler input.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a

geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and last characters.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

Step 2 Combine solutions with simpler inputs into a solution of the original problem.

In your mind, consider the solutions of your problem for the simpler inputs that you discovered in Step 1. Don't worry *how* those solutions are obtained. Simply have faith that the solutions are readily available. Just say to yourself: These are simpler inputs, so someone else will solve the problem for me.

Now think how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, related to the quantity that you lopped off to arrive at the simpler input. Maybe you cut the original input in half and have solutions for each half. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem a good idea. If you cut

"Madam, I'm Adam"

in half, you get two strings:

"Madam, I"

and

"'m Adam"

Neither of them is a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters. Removing the M at the front and the m at the back yields

"adam, I'm Ada"

Suppose you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match (ignoring letter case)
- and
- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

There is one other case to consider. What if the first or last letter of the word is not a letter? For example, the string

"A man, a plan, a canal, Panama!"

ends in a ! character, which does not match the A in the front. But we should ignore non-letters when testing for palindromes. Thus, when the last character is not a letter but the first character is a letter, it doesn't make sense to remove both the first and the last characters. That's not a problem. Remove only the last character. If the shorter string is a palindrome, then it stays a palindrome when you attach a nonletter.

The same argument applies if the first character is not a letter. Now we have a complete set of cases.

- If the first and last characters are both letters, then check whether they match. If so, remove both and test the shorter string.
- Otherwise, if the last character isn't a letter, remove it and test the shorter string.
- Otherwise, the first character isn't a letter. Remove it and test the shorter string.

In all three cases, you can use the solution to the simpler problem to arrive at a solution to your problem.

Step 3

Find solutions to the simplest inputs.

A recursive computation keeps simplifying its inputs. Eventually it arrives at very simple inputs. To make sure that the recursion comes to a stop, you must deal with the simplest inputs separately. Come up with special solutions for them, which is usually very easy.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Let's look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character
- The empty string

We don't need a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But we do need to worry about strings of length 0 and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string "mm". According to the rule discovered in Step 2, this string is a palindrome if the first and last characters of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

A string with a single letter, such as "I", is a palindrome. How about the case in which the character is not a letter, such as "?"? Removing the ! yields the empty string, which is a palindrome. Thus, we conclude that all strings of length 0 or 1 are palindromes.

Step 4

Implement the solution by combining the simple cases and the reduction step.

Now you are ready to implement the solution. Make separate cases for the simple inputs that you considered in Step 3. If the input isn't one of the simplest cases, then implement the logic you discovered in Step 2.

Here is the `isPalindrome` method:

```
public static boolean isPalindrome(String text)
{
    int length = text.length();

    // Separate case for shortest strings.
    if (length <= 1) { return true; }
    else
    {
        // Get first and last characters, converted to lowercase.
        char first = Character.toLowerCase(text.charAt(0));
        char last = Character.toLowerCase(text.charAt(length - 1));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {

```

```

        // Remove both first and last character.
        String shorter = text.substring(1, length - 1);
        return isPalindrome(shorter);
    }
    else
    {
        return false;
    }
}
else if (!Character.isLetter(last))
{
    // Remove last character.
    String shorter = text.substring(0, length - 1);
    return isPalindrome(shorter);
}
else
{
    // Remove first character.
    String shorter = text.substring(1);
    return isPalindrome(shorter);
}
}

```

FULL CODE EXAMPLE

Go to wiley.com/go/bj1o2code to download the complete Palindromes class.

WORKED EXAMPLE 13.1**Finding Files**

Learn how to use recursion to find all files with a given extension in a directory tree. Go to wiley.com/go/bj1o2examples and download Worked Example 13.1.



13.2 Recursive Helper Methods

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper method.

Here is a typical example: In the palindrome test of How To 13.1, it is a bit inefficient to construct new string objects in every step. Consider the following change in the problem: Instead of testing whether the entire sentence is a palindrome, let's check whether a substring is a palindrome:

```

/**
 * Tests whether a substring is a palindrome.
 * @param text a string that is being checked
 * @param start the index of the first character of the substring
 * @param end the index of the last character of the substring
 * @return true if the substring is a palindrome
 */
public static boolean isPalindrome(String text, int start, int end)

```



Sometimes, a task can be solved by handing it off to a recursive helper method.

© geremne/iStockphoto.

This method turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the start and end parameter variables to skip over matching letter pairs and characters that are not letters. There is no need to construct new String objects to represent the shorter strings.

```
public static boolean isPalindrome(String text, int start, int end)
{
    // Separate case for substrings of length 0 and 1.
    if (start >= end) { return true; }
    else
    {
        // Get first and last characters, converted to lowercase.
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            if (first == last)
            {
                // Test substring that doesn't contain the matching letters.
                return isPalindrome(text, start + 1, end - 1);
            }
            else
            {
                return false;
            }
        }
        else if (!Character.isLetter(last))
        {
            // Test substring that doesn't contain the last character.
            return isPalindrome(text, start, end - 1);
        }
        else
        {
            // Test substring that doesn't contain the first character.
            return isPalindrome(text, start + 1, end);
        }
    }
}
```

You should still supply a method to solve the whole problem—the user of your method shouldn't have to know about the trick with the substring positions. Simply call the helper method with positions that test the entire string:

```
public static boolean isPalindrome(String text)
{
    return isPalindrome(text, 0, text.length() - 1);
}
```



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download the `Palindromes` class with a helper method.

Note that this call is *not* a recursive method call. The `isPalindrome(String)` method calls the helper method `isPalindrome(String, int, int)`. In this example, we use **overloading** to declare two methods with the same name. The `isPalindrome` method with just a `String` parameter variable is the method that we expect the public to use. The second method, with one `String` and two `int` parameter variables, is the recursive helper method. If you prefer, you can avoid overloaded methods by choosing a different name for the helper method, such as `substringIsPalindrome`.

Use the technique of recursive helper methods whenever it is easier to solve a recursive problem that is equivalent to the original problem—but more amenable to a recursive solution.



6. Do we have to give the same name to both `isPalindrome` methods?
7. When does the recursive `isPalindrome` method stop calling itself?
8. To compute the sum of the values in an array, add the first value to the sum of the remaining values, computing recursively. Design a recursive helper method to solve this problem.
9. How can you write a recursive method `public static void sum(int[] a)` without needing a helper function? Why is this less efficient?

Practice It Now you can try these exercises at the end of the chapter: E13.6, E13.9, E13.13.

13.3 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool for implementing complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

Consider the Fibonacci sequence: a sequence of numbers defined by the equation

$$\begin{aligned}f_1 &= 1 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2}\end{aligned}$$



© pagadesign/Stockphoto.

In most cases, iterative and recursive approaches have comparable efficiency.

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is $34 + 55 = 89$.

We would like to write a method that computes f_n for any value of n . Here we translate the definition directly into a recursive method:

sec03/RecursiveFib.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program computes Fibonacci numbers using a recursive method.
5 */
6 public class RecursiveFib
7 {
8     public static void main(String[] args)
9     {
10        Scanner in = new Scanner(System.in);
11        System.out.print("Enter n: ");
12        int n = in.nextInt();

```

```

13
14     for (int i = 1; i <= n; i++)
15     {
16         long f = fib(i);
17         System.out.println("fib(" + i + ") = " + f);
18     }
19 }
20
21 /**
22  * Computes a Fibonacci number.
23  * @param n an integer
24  * @return the nth Fibonacci number
25 */
26 public static long fib(int n)
27 {
28     if (n <= 2) { return 1; }
29     else { return fib(n - 1) + fib(n - 2); }
30 }
31 }
```

Program Run

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

That is certainly simple, and the method will work correctly. But watch the output closely as you run the test program. The first few calls to the `fib` method are fast. For larger values, though, the program pauses an amazingly long time between outputs.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer anywhere near that long.

To find out the problem, let us insert **trace messages** into the method:

sec03/RecursiveFibTracer.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program prints trace messages that show how often the
5  * recursive method for computing Fibonacci numbers calls itself.
6 */
7 public class RecursiveFibTracer
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter n: ");
13        int n = in.nextInt();
14 }
```

```
15     long f = fib(n);
16
17     System.out.println("fib(" + n + ") = " + f);
18 }
19
20 /**
21  * Computes a Fibonacci number.
22  * @param n an integer
23  * @return the nth Fibonacci number
24 */
25 public static long fib(int n)
26 {
27     System.out.println("Entering fib: n = " + n);
28     long f;
29     if (n <= 2) { f = 1; }
30     else { f = fib(n - 1) + fib(n - 2); }
31     System.out.println("Exiting fib: n = " + n
32                         + " return value = " + f);
33     return f;
34 }
35 }
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

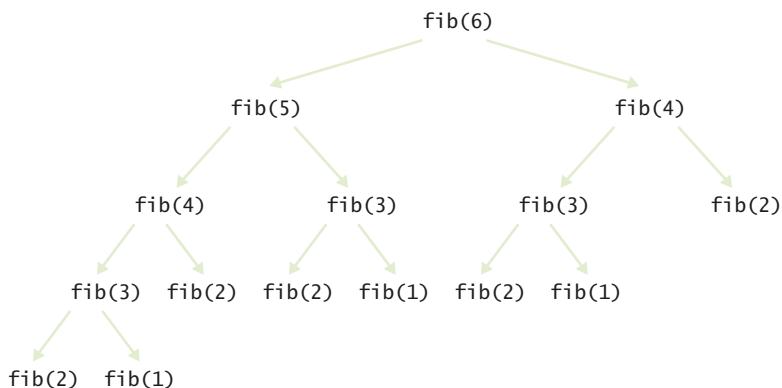


Figure 2 Call Pattern of the Recursive fib Method

Figure 2 shows the pattern of recursive calls for computing $\text{fib}(6)$. Now it is becoming apparent why the method takes so long. It is computing the same values over and over. For example, the computation of $\text{fib}(6)$ calls $\text{fib}(4)$ twice and $\text{fib}(3)$ three times. That is very different from the computation we would do with pencil and paper. There we would just write down the values as they were computed and add up the last two to get the next one until we reached the desired entry; no sequence value would ever be computed twice.

If we imitate the pencil-and-paper process, then we get the following program:

sec03/LoopFib.java

```
1 import java.util.Scanner;
2
3 /**
4  * This program computes Fibonacci numbers using an iterative method.
5 */
6 public class LoopFib
7 {
8     public static void main(String[] args)
9     {
10         Scanner in = new Scanner(System.in);
11         System.out.print("Enter n: ");
12         int n = in.nextInt();
13
14         for (int i = 1; i <= n; i++)
15         {
16             long f = fib(i);
17             System.out.println("fib(" + i + ") = " + f);
18         }
19     }
20
21     /**
22      * Computes a Fibonacci number.
23      * @param n an integer
24      * @return the nth Fibonacci number
25     */
26     public static long fib(int n)
27     {
28         if (n <= 2) { return 1; }
29         else
30         {
31             long a = 0, b = 1, c;
32             for (int i = 3; i <= n; i++)
33             {
34                 c = a + b;
35                 a = b;
36                 b = c;
37             }
38             return c;
39         }
40     }
41 }
```

```

31     long olderValue = 1;
32     long oldValue = 1;
33     long newValue = 1;
34     for (int i = 3; i <= n; i++)
35     {
36         newValue = oldValue + olderValue;
37         oldValue = oldValue;
38         oldValue = newValue;
39     }
40     return newValue;
41 }
42 }
43 }
```

Program Run

```

Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
...
fib(50) = 12586269025
```

This method runs *much* faster than the recursive version.

In this example of the `fib` method, the recursive solution was easy to program because it followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test:

```

public static boolean isPalindrome(String text)
{
    int start = 0;
    int end = text.length() - 1;
    while (start < end)
    {
        char first = Character.toLowerCase(text.charAt(start));
        char last = Character.toLowerCase(text.charAt(end));

        if (Character.isLetter(first) && Character.isLetter(last))
        {
            // Both are letters.
            if (first == last)
            {
                start++;
                end--;
            }
            else { return false; }
        }
        if (!Character.isLetter(last)) { end--; }
        if (!Character.isLetter(first)) { start++; }
    }
    return true;
}
```

Occasionally, a recursive solution is much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.

**FULL CODE EXAMPLE**

Go to wiley.com/go/bj102code to download the LoopPalindromes class.

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

SELF CHECK

10. Is it faster to compute the triangle numbers recursively, as shown in Section 13.1, or is it faster to use a loop that computes $1 + 2 + 3 + \dots + \text{width}$?
11. You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \dots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?
12. To compute the sum of the values in an array, you can split the array in the middle, recursively compute the sums of the halves, and add the results. Compare the performance of this algorithm with that of a loop that adds the values.

Practice It

Now you can try these exercises at the end of the chapter: R13.7, R13.9, E13.7, E13.27.

13.4 Permutations

The permutations of a string can be obtained more naturally through recursion than with a loop.

In this section, we will study a more complex example of recursion that would be difficult to program with a simple loop. (As Exercise P13.5 shows, it is possible to avoid the recursion, but the resulting solution is quite complex, and no faster).

We will design a method that lists all permutations of a string. A permutation is simply a rearrangement of the letters in the string. For example, the string "eat" has six permutations (including the original string itself):

"eat"	"ate"
"eta"	"tea"
"aet"	"tae"



© Jeanine Groenwald/
iStockphoto.

Using recursion, you can find all arrangements of a set of objects.

Now we need a way to generate the permutations recursively. Consider the string "eat". Let's simplify the problem. First, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? We need to know the permutations of the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Thus, we can use recursion. Generate the permutations of the substring "at". They are

```
"at"
"ta"
```

For each permutation of that substring, prepend the letter 'e' to get the permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```

Now let's turn our attention to the permutations of "eat" that start with 'a'. We need to produce the permutations of the remaining letters, "et". They are:

```
"et"
"te"
```

We add the letter 'a' to the front of the strings and obtain

```
"aet"
"ate"
```

We generate the permutations that start with 't' in the same way.

That's the idea. The implementation is fairly straightforward. In the `permutations` method, we loop through all positions in the word to be permuted. For each of them, we compute the shorter word that is obtained by removing the *i*th letter:

```
String shorter = word.substring(0, i) + word.substring(i + 1);
```

We compute the permutations of the shorter word:

```
ArrayList<String> shorterPermutations = permutations(shorter);
```

Finally, we add the removed letter to the front of all permutations of the shorter word.

```
for (String s : shorterPermutations)
{
    result.add(word.charAt(i) + s);
}
```

As always, we have to provide a special case for the simplest strings. The simplest possible string is the empty string, which has a single permutation—itself.

Here is the complete `Permutations` class:

sec04/Permutations.java

```
1 import java.util.ArrayList;
2
3 /**
4     This class computes permutations of a string.
5 */
6 public class Permutations
7 {
8     public static void main(String[] args)
9     {
10         for (String s : permutations("eat"))
11         {
```

```

12         System.out.println(s);
13     }
14 }
15 /**
16 * Gets all permutations of a given word.
17 * @param word the string to permute
18 * @return a list of all permutations
19 */
20 public static ArrayList<String> permutations(String word)
21 {
22     ArrayList<String> result = new ArrayList<String>();
23
24     // The empty string has a single permutation: itself
25     if (word.length() == 0)
26     {
27         result.add(word);
28         return result;
29     }
30     else
31     {
32         // Loop through all character positions
33         for (int i = 0; i < word.length(); i++)
34         {
35             // Form a shorter word by removing the ith character
36             String shorter = word.substring(0, i) + word.substring(i + 1);
37
38             // Generate all permutations of the simpler word
39             ArrayList<String> shorterPermutations = permutations(shorter)
40
41             // Add the removed character to the front of
42             // each permutation of the simpler word
43             for (String s : shorterPermutations)
44             {
45                 result.add(word.charAt(i) + s);
46             }
47         }
48         // Return all permutations
49         return result;
50     }
51 }
52 }
53 }
```

Program Run

```

eat
eta
aet
ate
tea
tae
```

Compare the `Permutations` and `Triangle` classes. Both of them work on the same principle. When they work on a more complex input, they first solve the problem for a simpler input. Then they combine the result for the simpler input with additional work to deliver the results for the more complex input. There really is no particular complexity behind that process as long as you think about the solution on that level only. However, behind the scenes, the simpler input creates even simpler input,

which creates yet another simplification, and so on, until one input is so simple that the result can be obtained without further help. It is interesting to think about this process, but it can also be confusing. What's important is that you can focus on the one level that matters—putting a solution together from the slightly simpler problem, ignoring the fact that the simpler problem also uses recursion to get its results.



Computing & Society 13.1 The Limits of Computation

Have you ever wondered how your instructor or grader makes sure your programming homework is correct? In all likelihood, they look at your solution and perhaps run it with some test inputs. But usually they have a correct solution available. That suggests that there might be an easier way. Perhaps they could feed your program and their correct program into a “program comparator”, a computer program that analyzes both programs and determines whether they both compute the same results. Of course, your solution and the program that is known to be correct need not be identical—what matters is that they produce the same output when given the same input.

How could such a program comparator work? Well, the Java compiler knows how to read a program and make sense of the classes, methods, and statements. So it seems plausible that someone could, with some effort, write a program that reads two Java programs, analyzes what they do, and determines whether they solve the same task. Of course, such a program would be very attractive to instructors, because it could automate the grading process. Thus, even though no such program exists today, it might be tempting to try to develop one and sell it to universities around the world.

However, before you start raising venture capital for such an effort, you should know that theoretical computer scientists have proven that it is impossible to develop such a program, *no matter how hard you try*.

There are quite a few of these unsolvable problems. The first one,

called the *halting problem*, was discovered by the British researcher Alan Turing in 1936. Because his research occurred before the first actual computer was constructed, Turing had to devise a theoretical device, the **Turing machine**, to explain how computers could work. The Turing machine consists of a long magnetic tape, a read/write head, and a program that has numbered instructions of the form: “If the current symbol under the head is *x*, then replace it with *y*, move the head one unit left or right, and continue with instruction *n*” (see the figure on the next page). Interestingly enough, with only these instructions, you can program just as much as with Java, even though it is incredibly tedious to do so. Theoretical computer scientists like Turing machines because they can be described using nothing more than the laws of mathematics.

Expressed in terms of Java, the halting problem states: “It is impossible to write a program with two inputs, namely the source code of an arbitrary Java program *P* and a string *I*, that decides whether the program *P*, when executed with the input *I*, will halt—that is, the program will not get into an infinite loop with the given input”. Of course, for some kinds of programs and inputs, it is possible to decide whether the program halts with the given input. The halting problem asserts that it is impossible to come up with a single decision-making algorithm that works with all programs and inputs. Note that you can't simply run the program *P* on the input *I* to settle this question. If the program runs for 1,000 days, you don't know that the

program is in an infinite loop. Maybe you just have to wait another day for it to stop.

Such a “halt checker”, if it could be written, might also be useful for grading homework. An instructor could use it to screen student submissions to see if they get into an infinite loop with a particular input, and then stop checking them. However, as Turing demonstrated, such a program cannot be written. His argument is ingenious and quite simple.

Suppose a “halt checker” program existed. Let's call it *H*. From *H*, we will develop another program, the “killer” program *K*. *K* does the following computation. Its input is a string containing the source code for a program *R*. It then applies the halt checker on the input program *R* and the input string *R*. That is, it checks whether the program *R* halts if its input is its own source code. It sounds bizarre to feed a program to itself, but it isn't impossible.



Science Photo Library/Photo Researchers.

Alan Turing

SELF CHECK

13. What are all permutations of the four-letter word beat?
14. Our recursion for the permutation generator stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?
15. Why isn't it easy to develop an iterative solution for the permutation generator?

Practice It Now you can try these exercises at the end of the chapter: E13.14, E13.15.

For example, the Java compiler is written in Java, and you can use it to compile itself. Or, as a simpler example, a word counting program can count the words in its own source code.

When K gets the answer from H that R halts when applied to itself, it is programmed to enter an infinite loop. Otherwise K exits. In Java, the program might look like this:

```
public class Killer
{
    public static void main(
        String[] args)
    {
        String r = read program input;
        HaltChecker checker =
            new HaltChecker();
        if (checker.check(r, r))
        {
            while (true)
                { // Infinite loop
                }
        }
        else
        {
            return;
        }
    }
}
```

Now ask yourself: What does the halt checker answer when asked whether K halts when given K as the input? Maybe it finds out that K gets into an infinite loop with such an input. But wait, that can't be right. That would mean that `checker.check(r, r)` returns false when `r` is the program code of K . As you can plainly see, in that case, the killer method returns, so K didn't get into an infinite loop. That shows that K must halt when analyzing itself, so

`checker.check(r, r)` should return true. But then the killer method doesn't terminate—it goes into an infinite loop. That shows that it is logically impossible to implement a program that can check whether *every* program halts on a particular input.

It is sobering to know that there are *limits* to computing. There are problems that no computer program, no matter how ingenious, can answer.

Theoretical computer scientists are working on other research involving the nature of computation. One important question that remains unsettled

to this day deals with problems that in practice are very time-consuming to solve. It may be that these problems are intrinsically hard, in which case it would be pointless to try to look for better algorithms. Such theoretical research can have important practical applications. For example, right now, nobody knows whether the most common encryption schemes used today could be broken by discovering a new algorithm. Knowing that no fast algorithms exist for breaking a particular code could make us feel more comfortable about the security of encryption.

Program

Instruction number	If tape symbol is	Replace with	Then move head	Then go to instruction
1	0	2	right	2
	1	1	left	4
2	0	0	right	2
	1	1	right	2
3	2	0	left	3
	0	0	left	3
4	1	1	left	3
	2	2	right	1
4	1	1	right	5
	2	0	left	4

Control unit

Read/write head



The Turing Machine

13.5 Mutual Recursion

In a mutual recursion, a set of cooperating methods calls each other repeatedly.

In the preceding examples, a method called itself to solve a simpler problem. Sometimes, a set of cooperating methods calls each other in a recursive fashion. In this section, we will explore such a **mutual recursion**. This technique is significantly more advanced than the simple recursion that we discussed in the preceding sections.

We will develop a program that can compute the values of arithmetic expressions such as

```
3+4*5
(3+4)*5
1-(2-(3-(4-5)))
```

Computing such an expression is complicated by the fact that * and / bind more strongly than + and -, and that parentheses can be used to group subexpressions.

Figure 3 shows a set of **syntax diagrams** that describes the syntax of these expressions. To see how the syntax diagrams work, consider the expression $3+4*5$:

- Enter the *expression* syntax diagram. The arrow points directly to *term*, giving you no alternative.
- Enter the *term* syntax diagram. The arrow points to *factor*, again giving you no choice.
- Enter the *factor* diagram. You have two choices: to follow the top branch or the bottom branch. Because the first input token is the number 3 and not a (, follow the bottom branch.
- Accept the input token because it matches the number. The unprocessed input is now $+4*5$.
- Follow the arrow out of *number* to the end of *factor*. As in a method call, you now back up, returning to the end of the *factor* element of the *term* diagram.
- Now you have another choice—to loop back in the *term* diagram, or to exit. The next input token is a +, and it matches neither the * nor the / that would be required to loop back. So you exit, returning to *expression*.

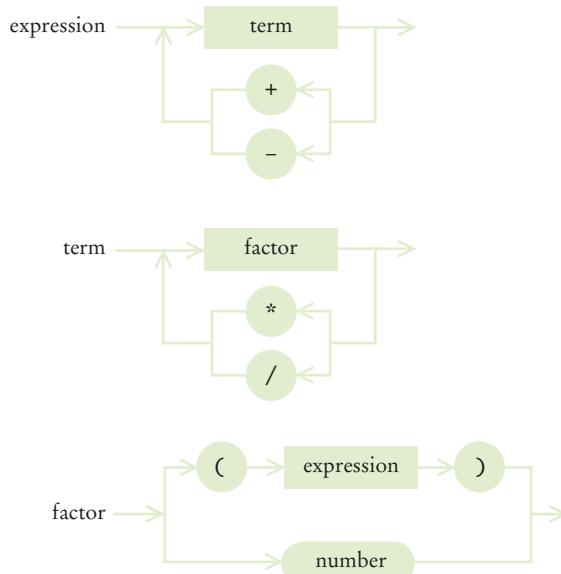


Figure 3
Syntax Diagrams for Evaluating an Expression

- Again, you have a choice, to loop back or to exit. Now the + matches one of the choices in the loop. Accept the + in the input and move back to the *term* element. The remaining input is $4*5$.

In this fashion, an expression is broken down into a sequence of terms, separated by + or -, each term is broken down into a sequence of factors, each separated by * or /, and each factor is either a parenthesized expression or a number. You can draw this breakdown as a tree. Figure 4 shows how the expressions $3+4*5$ and $(3+4)*5$ are derived from the syntax diagram.

Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

At the end of this section, you will find the implementation of the *Evaluator* class, which evaluates these expressions. The *Evaluator* makes use of an *ExpressionTokenizer* class, which breaks up an input string into **tokens**—numbers, operators, and parentheses. (For simplicity, we only accept positive integers as numbers, and we don't allow spaces in the input.)

When you call *nextToken*, the next input token is returned as a string. We also supply another method, *peekToken*, which allows you to see the next token without consuming it. To see why the *peekToken* method is necessary, consider the syntax diagram of the *term* type. If the next token is a "*" or "/", you want to continue adding and subtracting terms. But if the next token is another character, such as a "+" or "-", you want to stop without actually consuming it, so that the token can be considered later.

To compute the value of an expression, we implement three methods: *getExpressionValue*, *getTermValue*, and *getFactorValue*. The *getExpressionValue* method first calls *getTermValue* to get the value of the first term of the expression. Then it checks whether the next input token is one of + or -. If so, it calls *getTermValue* again and adds or subtracts it.

```
public int getExpressionValue()
{
    int value = getTermValue();
    boolean done = false;
    while (!done)
    {
        String next = tokenizer.peekToken();
        if ("+".equals(next) || "-".equals(next))
        {
```

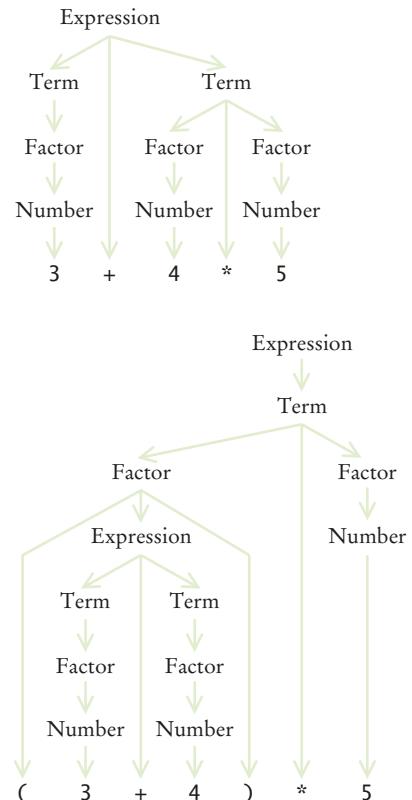


Figure 4
Syntax Trees for Two Expressions

```

        tokenizer.nextToken(); // Discard "+" or "-"
        int value2 = getTermValue();
        if ("+".equals(next)) { value = value + value2; }
        else { value = value - value2; }
    }
    else
    {
        done = true;
    }
}
return value;
}

```

The `getTermValue` method calls `getFactorValue` in the same way, multiplying or dividing the factor values.

Finally, the `getFactorValue` method checks whether the next input is a number, or whether it begins with a `(` token. In the first case, the value is simply the value of the number. However, in the second case, the `getFactorValue` method makes a recursive call to `getExpressionValue`. Thus, the three methods are mutually recursive.

```

public int getFactorValue()
{
    int value;
    String next = tokenizer.peekToken();
    if ("(".equals(next))
    {
        tokenizer.nextToken(); // Discard "("
        value = getExpressionValue();
        tokenizer.nextToken(); // Discard ")"
    }
    else
    {
        value = Integer.parseInt(tokenizer.nextToken());
    }
    return value;
}

```

To see the mutual recursion clearly, trace through the expression $(3+4)*5$:

- `getExpressionValue` calls `getTermValue`
 - `getTermValue` calls `getFactorValue`
 - `getFactorValue` consumes the `(` input
 - `getFactorValue` calls `getExpressionValue`
 - `getExpressionValue` returns eventually with the value of 7, having consumed $3 + 4$. This is the recursive call.
 - `getFactorValue` consumes the `)` input
 - `getFactorValue` returns 7
 - `getTermValue` consumes the inputs `*` and `5` and returns 35
 - `getExpressionValue` returns 35

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see when you consider the situation in which `getExpressionValue` calls itself. The second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the tokens of the input string are consumed, so eventually the recursion must come to an end.

sec05/Evaluator.java

```

1  /**
2   * A class that can compute the value of an arithmetic expression.
3  */
4  public class Evaluator
5  {
6      private ExpressionTokenizer tokenizer;
7
8      /**
9       * Constructs an evaluator.
10      * @param anExpression a string containing the expression
11      * to be evaluated
12     */
13    public Evaluator(String anExpression)
14    {
15        tokenizer = new ExpressionTokenizer(anExpression);
16    }
17
18    /**
19     * Evaluates the expression.
20     * @return the value of the expression
21    */
22    public int getExpressionValue()
23    {
24        int value = getTermValue();
25        boolean done = false;
26        while (!done)
27        {
28            String next = tokenizer.peekToken();
29            if ("+".equals(next) || "-".equals(next))
30            {
31                tokenizer.nextToken(); // Discard "+" or "-"
32                int value2 = getTermValue();
33                if ("+".equals(next)) { value = value + value2; }
34                else { value = value - value2; }
35            }
36            else
37            {
38                done = true;
39            }
40        }
41        return value;
42    }
43
44    /**
45     * Evaluates the next term found in the expression.
46     * @return the value of the term
47    */
48    public int getTermValue()
49    {
50        int value = getFactorValue();
51        boolean done = false;
52        while (!done)
53        {
54            String next = tokenizer.peekToken();
55            if ("*".equals(next) || "/".equals(next))
56            {
57                tokenizer.nextToken();
58                int value2 = getFactorValue();

```

```

59         if ("*".equals(next)) { value = value * value2; }
60     } else { value = value / value2; }
61   }
62   else
63   {
64     done = true;
65   }
66 }
67 return value;
68 }
69 /**
70 Evaluates the next factor found in the expression.
71 @return the value of the factor
72 */
73 public int getFactorValue()
74 {
75   int value;
76   String next = tokenizer.peekToken();
77   if ("(".equals(next))
78   {
79     tokenizer.nextToken(); // Discard "("
80     value = getExpressionValue();
81     tokenizer.nextToken(); // Discard ")"
82   }
83   else
84   {
85     value = Integer.parseInt(tokenizer.nextToken());
86   }
87   return value;
88 }
89 }
90 }
```

sec05/ExpressionTokenizer.java

```

1  /**
2   * This class breaks up a string describing an expression
3   * into tokens: numbers, parentheses, and operators.
4   */
5  public class ExpressionTokenizer
6  {
7    private String input;
8    private int start; // The start of the current token
9    private int end; // The position after the end of the current token
10
11 /**
12  * Constructs a tokenizer.
13  * @param anInput the string to tokenize
14  */
15 public ExpressionTokenizer(String anInput)
16 {
17   input = anInput;
18   start = 0;
19   end = 0;
20   nextToken(); // Find the first token
21 }
22
23 /**
24 * Peeks at the next token without consuming it.
```

```

25     @return the next token or null if there are no more tokens
26 */
27 public String peekToken()
28 {
29     if (start >= input.length()) { return null; }
30     else { return input.substring(start, end); }
31 }
32
33 /**
34  * Gets the next token and moves the tokenizer to the following token.
35  * @return the next token or null if there are no more tokens
36 */
37 public String nextToken()
38 {
39     String r = peekToken();
40     start = end;
41     if (start >= input.length()) { return r; }
42     if (Character.isDigit(input.charAt(start)))
43     {
44         end = start + 1;
45         while (end < input.length()
46                 && Character.isDigit(input.charAt(end)))
47         {
48             end++;
49         }
50     }
51     else
52     {
53         end = start + 1;
54     }
55     return r;
56 }
57 }
```

sec05/ExpressionCalculator.java

```

1 import java.util.Scanner;
2
3 /**
4  * This program calculates the value of an expression
5  * consisting of numbers, arithmetic operators, and parentheses.
6 */
7 public class ExpressionCalculator
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12        System.out.print("Enter an expression: ");
13        String input = in.nextLine();
14        Evaluator e = new Evaluator(input);
15        int value = e.getExpressionValue();
16        System.out.println(input + " = " + value);
17    }
18 }
```

Program Run

Enter an expression: 3+4*5
3+4*5=23



- 16.** What is the difference between a term and a factor? Why do we need both concepts?
- 17.** Why does the expression evaluator use mutual recursion?
- 18.** What happens if you try to evaluate the illegal expression $3+4^*)5$? Specifically, which method throws an exception?

Practice It Now you can try these exercises at the end of the chapter: R13.13, E13.21.

13.6 Backtracking

Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

Backtracking is a problem solving technique that builds up partial solutions that get increasingly closer to the goal. If a partial solution cannot be completed, one abandons it and returns to examining the other candidates.

Backtracking can be used to solve crossword puzzles, escape from mazes, or find solutions to systems that are constrained by rules. In order to employ backtracking for a particular problem, we need two characteristic properties:

- 1.** A procedure to examine a partial solution and determine whether to
 - Accept it as an actual solution.
 - Abandon it (either because it violates some rules or because it is clear that it can never lead to a valid solution).
 - Continue extending it.
- 2.** A procedure to extend a partial solution, generating one or more solutions that come closer to the goal.

Backtracking can then be expressed with the following recursive algorithm:

```

Solve(partialSolution)
  Examine(partialSolution).
  If accepted
    Add partialSolution to the list of solutions.
  Else if continuing
    For each p in extend(partialSolution)
      Solve(p).
  
```

Of course, the processes of examining and extending a partial solution depend on the nature of the problem.



© Lanica Klein/iStockphoto.

In a backtracking algorithm, one explores all paths toward a solution. When one path is a dead end, one needs to backtrack and try another choice.

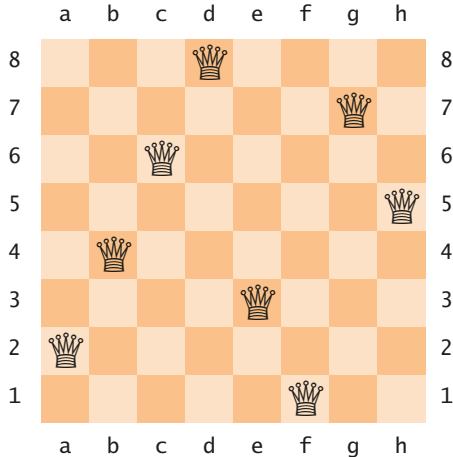


Figure 5 A Solution to the Eight Queens Problem

As an example, we will develop a program that finds all solutions to the eight queens problem: the task of positioning eight queens on a chess board so that none of them attacks another according to the rules of chess. In other words, there are no two queens on the same row, column, or diagonal. Figure 5 shows a solution.

In this problem, it is easy to examine a partial solution. If two queens attack another, reject it. Otherwise, if it has eight queens, accept it. Otherwise, continue.

It is also easy to extend a partial solution. Simply add another queen on an empty square.

However, in the interest of efficiency, we will be a bit more systematic about the extension process. We will place the first queen in row 1, the next queen in row 2, and so on.

We provide a class `PartialSolution` that collects the queens in a partial solution, and that has methods to examine and extend the solution:

```
public class PartialSolution
{
    private Queen[] queens;

    public int examine() { . . . }
    public PartialSolution[] extend() { . . . }
}
```

The `examine` method simply checks whether two queens attack each other:

```
public int examine()
{
    for (int i = 0; i < queens.length; i++)
    {
        for (int j = i + 1; j < queens.length; j++)
        {
            if (queens[i].attacks(queens[j])) { return ABANDON; }
        }
    }
    if (queens.length == NQUEENS) { return ACCEPT; }
    else { return CONTINUE; }
}
```

The extend method takes a given solution and makes eight copies of it. Each copy gets a new queen in a different column.

```
public PartialSolution[] extend()
{
    // Generate a new solution for each column
    PartialSolution[] result = new PartialSolution[NQUEENS];
    for (int i = 0; i < result.length; i++)
    {
        int size = queens.length;

        // The new solution has one more row than this one
        result[i] = new PartialSolution(size + 1);

        // Copy this solution into the new one
        for (int j = 0; j < size; j++)
        {
            result[i].queens[j] = queens[j];
        }

        // Append the new queen into the ith column
        result[i].queens[size] = new Queen(size, i);
    }
    return result;
}
```

You will find the Queen class at the end of the section. The only challenge is to determine when two queens attack each other diagonally. Here is an easy way of checking that. Compute the slope and check whether it is ± 1 . This condition can be simplified as follows:

$$\begin{aligned} (\text{row}_2 - \text{row}_1)/(\text{column}_2 - \text{column}_1) &= \pm 1 \\ \text{row}_2 - \text{row}_1 &= \pm(\text{column}_2 - \text{column}_1) \\ |\text{row}_2 - \text{row}_1| &= |\text{column}_2 - \text{column}_1| \end{aligned}$$

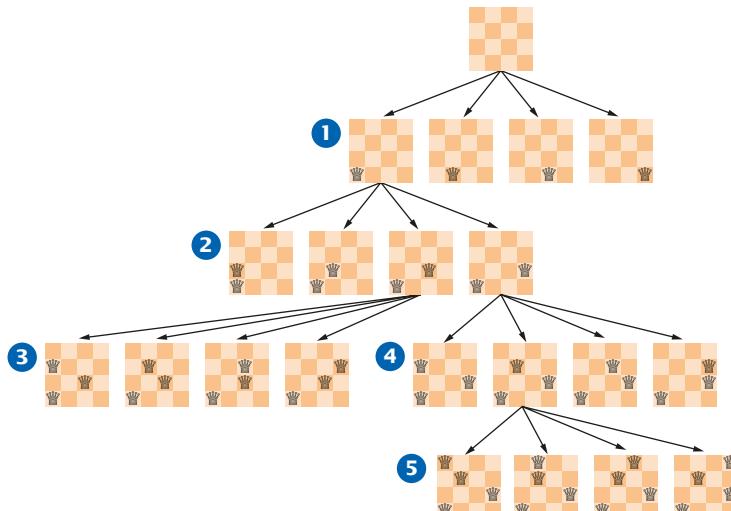


Figure 6 Backtracking in the Four Queens Problem

Have a close look at the `solve` method in the `EightQueens` class on page 639. The method is a straightforward translation of the pseudocode for backtracking. Note how there is nothing specific about the eight queens problem in this method—it works for any partial solution with an `examine` and `extend` method (see Exercise E13.22).

Figure 6 shows the `solve` method in action for a four queens problem. Starting from a blank board, there are four partial solutions with a queen in row 1 ①. When the queen is in column 1, there are four partial solutions with a queen in row 2 ②. Two of them are abandoned immediately. The other two lead to partial solutions with three queens ③ and ④, all but one of which are abandoned. One partial solution is extended to four queens, but all of those are abandoned as well ⑤. Then the algorithm backtracks, giving up on a queen in position a1, instead extending the solution with the queen in position b1 (not shown).

When you run the program, it lists 92 solutions, including the one in Figure 5. Exercise E13.23 asks you to remove those that are rotations or reflections of another.

sec06/PartialSolution.java

```

1 import java.util.Arrays;
2
3 /**
4  * A partial solution to the eight queens puzzle.
5 */
6 public class PartialSolution
7 {
8     private Queen[] queens;
9     private static final int NQUEENS = 8;
10
11    public static final int ACCEPT = 1;
12    public static final int ABANDON = 2;
13    public static final int CONTINUE = 3;
14
15    /**
16     * Constructs a partial solution of a given size.
17     * @param size the size
18     */
19    public PartialSolution(int size)
20    {
21        queens = new Queen[size];
22    }
23
24    /**
25     * Examines a partial solution.
26     * @return one of ACCEPT, ABANDON, CONTINUE
27     */
28    public int examine()
29    {
30        for (int i = 0; i < queens.length; i++)
31        {
32            for (int j = i + 1; j < queens.length; j++)
33            {
34                if (queens[i].attacks(queens[j])) { return ABANDON; }
35            }
36        }
37        if (queens.length == NQUEENS) { return ACCEPT; }
38        else { return CONTINUE; }
39    }
40

```

```

41  /**
42   * Yields all extensions of this partial solution.
43   * @return an array of partial solutions that extend this solution.
44   */
45  public PartialSolution[] extend()
46  {
47      // Generate a new solution for each column
48      PartialSolution[] result = new PartialSolution[NQUEENS];
49      for (int i = 0; i < result.length; i++)
50      {
51          int size = queens.length;
52
53          // The new solution has one more row than this one
54          result[i] = new PartialSolution(size + 1);
55
56          // Copy this solution into the new one
57          for (int j = 0; j < size; j++)
58          {
59              result[i].queens[j] = queens[j];
60          }
61
62          // Append the new queen into the ith column
63          result[i].queens[size] = new Queen(size, i);
64      }
65      return result;
66  }
67
68  public String toString() { return Arrays.toString(queens); }
69 }

```

sec06/Queen.java

```

1  /**
2   * A queen in the eight queens problem.
3   */
4  public class Queen
5  {
6      private int row;
7      private int column;
8
9      /**
10       * Constructs a queen at a given position.
11       * @param r the row
12       * @param c the column
13     */
14     public Queen(int r, int c)
15     {
16         row = r;
17         column = c;
18     }
19
20    /**
21     * Checks whether this queen attacks another.
22     * @param other the other queen
23     * @return true if this and the other queen are in the same
24     *         row, column, or diagonal
25     */
26     public boolean attacks(Queen other)
27     {

```

```

28     return row == other.row
29     || column == other.column
30     || Math.abs(row - other.row) == Math.abs(column - other.column);
31 }
32
33 public String toString()
34 {
35     return "" + "abcdefgh".charAt(column) + (row + 1) ;
36 }
37 }
```

sec06/EightQueens.java

```

1  /**
2   * This class solves the eight queens problem using backtracking.
3   */
4  public class EightQueens
5  {
6      public static void main(String[] args)
7      {
8          solve(new PartialSolution(0));
9      }
10
11 /**
12  * Prints all solutions to the problem that can be extended from
13  * a given partial solution.
14  * @param sol the partial solution
15  */
16  public static void solve(PartialSolution sol)
17  {
18      int exam = sol.examine();
19      if (exam == PartialSolution.ACCEPT)
20      {
21          System.out.println(sol);
22      }
23      else if (exam == PartialSolution.CONTINUE)
24      {
25          for (PartialSolution p : sol.extend())
26          {
27              solve(p);
28          }
29      }
30  }
31 }
```

Program Run

```

[a1, e2, h3, f4, c5, g6, b7, d8]
[a1, f2, h3, c4, g5, d6, b7, e8]
[a1, g2, d3, f4, h5, b6, e7, c8]
. .
[f1, a2, e3, b4, h5, c6, g7, d8]
. .
[h1, c2, a3, f4, b5, e6, g7, d8]
[h1, d2, a3, c4, f5, b6, g7, e8]
(92 solutions)
```



19. Why does j begin at $i + 1$ in the `examine` method?
20. Continue tracing the four queens problem as shown in Figure 6. How many solutions are there with the first queen in position a_2 ?
21. How many solutions are there altogether for the four queens problem?

Practice It Now you can try these exercises at the end of the chapter: E13.22, E13.25, E13.26.

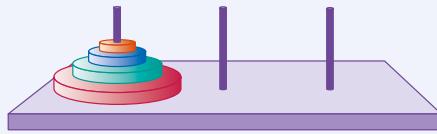


WORKED EXAMPLE 13.2

Towers of Hanoi



No discussion of recursion would be complete without the “Towers of Hanoi”. Learn how to solve this classic puzzle with an elegant recursive solution. Go to wiley.com/go/bjlo2examples and download Worked Example 13.2.



CHAPTER SUMMARY

Understand the control flow in a recursive computation.



- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest values.



Identify recursive helper methods for solving a problem.



- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Contrast the efficiency of recursive and non-recursive algorithms.



- Occasionally, a recursive solution is much slower than its iterative counterpart. However, in most cases, the recursive solution is only slightly slower.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Review a complex recursion example that cannot be solved with a simple loop.

- The permutations of a string can be obtained more naturally through recursion than with a loop.



Recognize the phenomenon of mutual recursion in an expression evaluator.

- In a mutual recursion, a set of cooperating methods calls each other repeatedly.

Use backtracking to solve problems that require trying out multiple paths.

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates.

**REVIEW EXERCISES**

- **R13.1** Define the terms
 - a. Recursion
 - b. Iteration
 - c. Infinite recursion
 - d. Recursive helper method

- **R13.2** Outline, but do not implement, a recursive solution for finding the smallest value in an array.

- **R13.3** Outline, but do not implement, a recursive solution for finding the k th smallest element in an array. *Hint:* Look at the elements that are less than the initial element. Suppose there are m of them. How should you proceed if $k \leq m$? If $k > m$?

- **R13.4** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.

- **R13.5** Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First sort the subarray without the initial element.

- **R13.6** Write a recursive definition of x^n , where $n \geq 0$, similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute x^n from x^{n-1} ? How does the recursion terminate?

- **R13.7** Improve upon Exercise R13.6 by computing x^n as $(x^{n/2})^2$ if n is even. Why is this approach significantly faster? *Hint:* Compute x^{1023} and x^{1024} both ways.

- **R13.8** Write a recursive definition of $n! = 1 \times 2 \times \dots \times n$, similar to the recursive definition of the Fibonacci numbers.

- **R13.9** Find out how often the recursive version of `fib` calls itself. Keep a static variable `fibCount` and increment it once in every call to `fib`. What is the relationship between `fib(n)` and `fibCount`?

- **R13.10** Let $\text{moves}(n)$ be the number of moves required to solve the Towers of Hanoi problem (see Worked Example 13.2). Find a formula that expresses $\text{moves}(n)$ in terms of $\text{moves}(n-1)$. Then show that $\text{moves}(n) = 2^n - 1$.

- **R13.11** Outline, but do not implement, a recursive solution for generating all subsets of the set $\{1, 2, \dots, n\}$.

- **R13.12** Exercise P13.5 shows an iterative way of generating all permutations of the sequence $(0, 1, \dots, n-1)$. Explain why the algorithm produces the correct result.

- **R13.13** Trace the expression evaluator program from Section 13.5 with inputs $3 - 4 + 5$, $3 - (4 + 5)$, $(3 - 4) * 5$, and $3 * 4 + 5 * 6$.

PRACTICE EXERCISES

- **E13.1** Given a class `Rectangle` with instance variables `width` and `height`, provide a recursive `getArea` method. Construct a rectangle whose width is one less than the original and call its `getArea` method.
- ■ **E13.2** Given a class `Square` with an instance variable `width`, provide a recursive `getArea` method. Construct a square whose width is one less than the original and call its `getArea` method.
- ■ **E13.3** Write a recursive method for factoring an integer n . First, find a factor f , then recursively factor n / f .
- ■ **E13.4** Write a recursive method for computing a string with the binary digits of a number. If n is even, then the last digit is 0. If n is odd, then the last digit is 1. Recursively obtain the remaining digits.
- ■ **E13.5** Write a recursive method `String reverse(String text)` that reverses a string. For example, `reverse("Hello!")` returns the string "`!olleH`". Implement a recursive solution by removing the first character, reversing the remaining text, and combining the two.
- ■ ■ **E13.6** Redo Exercise E13.5 with a recursive helper method that reverses a substring of the message text.
- ■ ■ **E13.7** Implement the `reverse` method of Exercise E13.5 as an iteration.
- ■ ■ **E13.8** Use recursion to implement a method

```
public static boolean find(String text, String str)
```

that tests whether a given text contains a string. For example, `find("Mississippi", "sip")` returns true.

Hint: If the text starts with the string you want to match, then you are done. If not, consider the text that you obtain by removing the first character.

- ■ ■ **E13.9** Use recursion to implement a method

```
public static int indexOf(String text, String str)
```

that returns the starting position of the first substring of the text that matches `str`. Return `-1` if `str` is not a substring of the text.

For example, `s.indexOf("Mississippi", "sip")` returns 6.

Hint: This is a bit trickier than Exercise E13.8, because you must keep track of how far the match is from the beginning of the text. Make that value a parameter variable of a helper method.

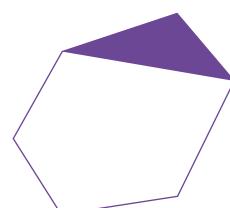
- ■ ■ **E13.10** Using recursion, find the largest element in an array.

Hint: Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

- ■ ■ **E13.11** Using recursion, compute the sum of all values in an array.

- ■ ■ **E13.12** Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners (x_1, y_1) , (x_2, y_2) , (x_3, y_3) has area

$$\frac{|x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1|}{2}$$



- ... E13.13** The following method was known to the ancient Greeks for computing square roots. Given a value $x > 0$ and a guess g for the square root, a better guess is $(g + x/g) / 2$. Write a recursive helper method `public static squareRootGuess(double x, double g)`. If g^2 is approximately equal to x , return g , otherwise, return `squareRootGuess` with the better guess. Then write a method `public static squareRoot(double x)` that uses the helper method.
- ... E13.14** Implement a `SubstringGenerator` that generates all substrings of a string. For example, the substrings of the string "rum" are the seven strings
- ```
"r", "ru", "rum", "u", "um", "m", ""
```
- Hint:* First enumerate all substrings that start with the first character. There are  $n$  of them if the string has length  $n$ . Then enumerate the substrings of the string that you obtain by removing the first character.
- ... E13.15** Implement a `SubsetGenerator` that generates all subsets of the characters of a string. For example, the subsets of the characters of the string "rum" are the eight strings
- ```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```
- Note that the subsets don't have to be substrings—for example, "rm" isn't a substring of "rum".
- ... E13.16** Recursively generate all ways in which an array list can be split up into a sequence of nonempty sublists. For example, if you are given the array list [1, 7, 2, 9], return the following lists of lists:
- ```
[[[1], [7], [2], [9]], [[1, 7], [2], [9]], [[1], [7, 2], [9]], [[1, 7, 2], [9]],
[[1], [7], [2, 9]], [[1, 7], [2, 9]], [[1], [7, 2, 9]], [[1, 7, 2, 9]]]
```
- Hint:* First generate all sublists of the list with the last element removed. The last element can either be a subsequence of length 1, or it can be added to the last subsequence.
- ... E13.17** Given an array list `a` of integers, recursively find all lists of elements of `a` whose sum is a given integer `n`.
- ... E13.18** Suppose you want to climb a staircase with  $n$  steps and you can take either one or two steps at a time. Recursively enumerate all paths. For example, if  $n$  is 5, the possible paths are:
- ```
[1, 2, 3, 4, 5], [1, 3, 4, 5], [1, 2, 4, 5], [1, 2, 3, 5], [1, 4, 5]
```
- ... E13.19** Repeat Exercise E13.18, where the climber can take up to k steps at a time.
- ... E13.20** Given an integer `price`, list all possible ways of paying for it with \$100, \$20, \$5, and \$1 bills, using recursion. Don't list duplicates.
- ... E13.21** Extend the expression evaluator in Section 13.5 so that it can handle the `%` operator as well as a “raise to a power” operator `^`. For example, 2^3 should evaluate to 8. As in mathematics, raising to a power should bind more strongly than multiplication: $5 * 2^3$ is 40.
- ... E13.22** The backtracking algorithm will work for any problem whose partial solutions can be examined and extended. Provide a `PartialSolution` interface type with methods `examine` and `extend`, a `solve` method that works with this interface type, and a class `EightQueensPartialSolution` that implements the interface.

- **E13.23** Refine the program for solving the eight queens problem so that rotations and reflections of previously displayed solutions are not shown. Your program should display twelve unique solutions.
- **E13.24** Refine the program for solving the eight queens problem so that the solutions are written to an HTML file, using tables with black and white background for the board and the Unicode character ☜ '\u2655' for the white queen.
- **E13.25** Generalize the program for solving the eight queens problem to the n queens problem. Your program should prompt for the value of n and display the solutions.
- **E13.26** Using backtracking, write a program that solves summation puzzles in which each letter should be replaced by a digit, such as

send + more = money

Other examples are base + ball = games and kyoto + osaka = tokyo.

- **E13.27** The recursive computation of Fibonacci numbers can be speeded up significantly by keeping track of the values that have already been computed. Provide an implementation of the `fib` method that uses this strategy. Whenever you return a new value, also store it in an auxiliary array. However, before embarking on a computation, consult the array to find whether the result has already been computed. Compare the running time of your improved implementation with that of the original recursive implementation and the loop implementation.

PROGRAMMING PROJECTS

- **P13.1** Phone numbers and PIN codes can be easier to remember when you find words that spell out the number on a standard phone pad. For example, instead of remembering the combination 5282, you can just think of JAVA.
Write a recursive method that, given a number, yields all possible spellings (which may or may not be real words).
- **P13.2** Continue Exercise P13.1, checking the words against the `/usr/share/dict/words` file on your computer, or the `words.txt` file in the companion code for this book. For a given number, return only actual words.
- **P13.3** With a longer number, you may need more than one word to remember it on a phone pad. For example, 263-346-5282 is CODE IN JAVA. Using your work from Exercise P13.2, write a program that, given any number, lists all word sequences that spell the number on a phone pad.
- **P13.4** Change the `permutations` method of Section 13.4 (which computed all permutations at once) to a `PermutationIterator` (which computes them one at a time).

```
public class PermutationIterator
{
    public PermutationIterator(String s) { . . . }
    public String nextPermutation() { . . . }
    public boolean hasMorePermutations() { . . . }
}
```

Here is how you would print out all permutations of the string "eat":

```
PermutationIterator iter = new PermutationIterator("eat");
while (iter.hasMorePermutations())
{
```

```

        System.out.println(iter.nextPermutation());
    }
}

```

Now we need a way to iterate through the permutations recursively. Consider the string "eat". As before, we'll generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do we generate the permutations that start with 'e'? Make another `PermutationIterator` object (called `tailIterator`) that iterates through the permutations of the substring "at". In the `nextPermutation` method, simply ask `tailIterator` what its next permutation is, and then add the 'e' at the front. However, there is one special case. When the tail generator runs out of permutations, all permutations that start with the current letter have been enumerated. Then

- Increment the current position.
- Compute the tail string that contains all letters except for the current one.
- Make a new permutation iterator for the tail string.

You are done when the current position has reached the end of the string.

••• P13.5 The following class generates all permutations of the numbers $0, 1, 2, \dots, n - 1$, without using recursion.

```

public class NumberPermutationIterator
{
    private int[] a;

    public NumberPermutationIterator(int n)
    {
        a = new int[n];
        done = false;
        for (int i = 0; i < n; i++) { a[i] = i; }
    }

    public int[] nextPermutation()
    {
        if (a.length <= 1) { return a; }

        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i])
            {
                int j = a.length - 1;
                while (a[i - 1] > a[j]) { j--; }
                swap(i - 1, j);
                reverse(i, a.length - 1);
                return a;
            }
        }
        return a;
    }

    public boolean hasMorePermutations()
    {
        if (a.length <= 1) { return false; }
        for (int i = a.length - 1; i > 0; i--)
        {
            if (a[i - 1] < a[i]) { return true; }
        }
        return false;
    }
}

```

```

public void swap(int i, int j)
{
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

public void reverse(int i, int j)
{
    while (i < j) { swap(i, j); i++; j--; }
}
}

```

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this class to get all permutations of the character positions and then compute a string whose i th character is `word.charAt(a[i])`. Use this approach to reimplement the `PermutationIterator` of Exercise P13.4 without recursion.

- P13.6** Implement an iterator that produces the moves for the Towers of Hanoi puzzle described in Worked Example 13.2. Provide methods `hasMoreMoves` and `nextMove`. The `nextMove` method should yield a string describing the next move. For example, the following code prints all moves needed to move five disks from peg 1 to peg 3:

```

DiskMover mover = new DiskMover(5, 1, 3);
while (mover.hasMoreMoves())
{
    System.out.println(mover.nextMove());
}

```

Hint: A disk mover that moves a single disk from one peg to another simply has a `nextMove` method that returns a string

Move disk from peg *source* to *target*

A disk mover with more than one disk to move must work harder. It needs another `DiskMover` to help it move the first $d - 1$ disks. Then `nextMove` asks that disk mover for its next move until it is done. Then the `nextMove` method issues a command to move the d th disk. Finally, it constructs another disk mover that generates the remaining moves.

It helps to keep track of the state of the disk mover:

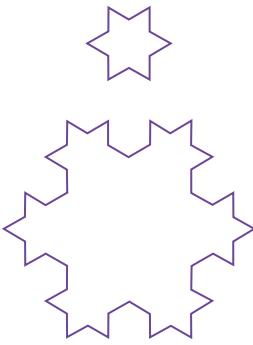
- BEFORE_LARGEST: A helper mover moves the smaller pile to the other peg.
- LARGEST: Move the largest disk from the source to the destination.
- AFTER_LARGEST: The helper mover moves the smaller pile from the other peg to the target.
- DONE: All moves are done.

- P13.7** *Escaping a Maze.* You are currently located inside a maze. The walls of the maze are indicated by asterisks (*).

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return true. Recursively check whether you can escape from one of the empty neighboring locations without visiting the current location. This method merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.



- P13.8** Using the `PartialSolution` interface and `solve` method from Exercise E13.22, provide a class `MazePartialSolution` for solving the maze escape problem of Exercise P13.7.

- ... P13.9** The expression evaluator in Section 13.5 returns the value of an expression. Modify the evaluator so that it returns an instance of the Expression interface with five implementing classes, Constant, Sum, Difference, Product, and Quotient. The Expression interface has a method `int value()`. The Constant class stores a number, which is returned by the `value` method. The other four classes store two arguments of type Expression, and their `value` method returns the sum, difference, product, and quotient of the values of the arguments. Write a test program that reads an expression string, translates it into an Expression object, and prints the result of calling `value`.
- ... P13.10** Refine the expression evaluator of Exercise P13.9 so that expressions can contain the variable `x`. For example, $3*x*x+4*x+5$ is a valid expression. Change the Expression interface so that its `value` method has as parameter the value that `x` should take. Add a class Variable that denotes an `x`. Write a program that reads an expression string and a value for `x`, translates the expression string into an Expression object, and prints the result of calling `value(x)`.
- ... P13.11** Add a `toString` method to the Expression class (as described in Exercise P13.9 and Exercise P13.10) that returns a string representation of the expression. It is ok to use more parentheses than required in mathematical notation. For example, for the expression $3*x*x+5$, you can print $((3*x)*x)+5$.
- ... P13.12** Write a program that reads an expression involving integers and the variable `x` into an Expression object, and then computes the derivative. Add a method `Expression derivative()` to the Expression interface. Use the rules from calculus for computing the derivative of a sum, difference, product, quotient, constant, or variable. Don't simplify the result. Print the resulting expression. For example, when reading `x * x`, you should print $((1*x)+(x*1))$.
- ... Graphics P13.13** *The Koch Snowflake.* A snowflake-like shape is recursively defined as follows. Start with an equilateral triangle:
- 
- Next, increase the size by a factor of three and replace each straight line with four line segments:
- 
- Repeat the process:
- Write a program that draws the iterations of the snowflake shape. Supply a button that, when clicked, produces the next iteration.

ANSWERS TO SELF-CHECK QUESTIONS

- Suppose we omit the statement. When computing the area of a triangle with width 1, we compute the area of the triangle with width 0 as 0, and then add 1 to arrive at the correct area.

2. You would compute the smaller area recursively, then return

```
smallerArea + width + width - 1.
```



Of course, it would be simpler to compute the area simply as $\text{width} * \text{width}$. The results are identical because

$$1 + 0 + 2 + 1 + 3 + 2 + \dots + n + n - 1 = \\ \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2$$

3. There is no provision for stopping the recursion. When a number < 10 isn't 8, then the method should return false and stop.

```
4. public static int pow2(int n)
{
    if (n <= 0) { return 1; } // 20 is 1
    else { return 2 * pow2(n - 1); }
}
```

```
5. mystery(4) calls mystery(3)
mystery(3) calls mystery(2)
mystery(2) calls mystery(1)
mystery(1) calls mystery(0)
mystery(0) returns 0.
mystery(1) returns 0 + 1 * 1 = 1
mystery(2) returns 1 + 2 * 2 = 5
mystery(3) returns 5 + 3 * 3 = 14
mystery(4) returns 14 + 4 * 4 = 30
```

6. No—the second one could be given a different name such as `substringIsPalindrome`.
7. When `start >= end`, that is, when the investigated string is either empty or has length 1.
8. The method `sumHelper(int[] a, int start)` adds `a[start]` and `sumHelper(a, start + 1)`.
9. `sum(a)` can make a new array `smaller` containing `a[1] ... a[a.length - 1]` and compute `a[0] + sum(smaller)`. But it is inefficient to make a copy of the array in each step.
10. The loop is slightly faster. It is even faster to simply compute $\text{width} * (\text{width} + 1) / 2$.
11. No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.

12. The recursive algorithm performs about as well as the loop. Unlike the recursive Fibonacci algorithm, this algorithm doesn't call itself again on the same input. For example, the sum of the array 1 4 9 16 25 36 49 64 is computed as the sum of 1 4 9 16 and 25 36 49 64, then as the sums of 1 4, 9 16, 25 36, and 49 64, which can be computed directly.

13. They are b followed by the six permutations of eat, e followed by the six permutations of bat, a followed by the six permutations of bet, and t followed by the six permutations of bea.

14. Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

15. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations eat, eta, and aet, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious—see Exercise P13.5.

16. Factors are combined by multiplicative operators (* and /); terms are combined by additive operators (+, -). We need both so that multiplication can bind more strongly than addition.

17. To handle parenthesized expressions, such as $2+3*(4+5)$. The subexpression $4+5$ is handled by a recursive call to `getExpressionValue`.

18. The `Integer.parseInt` call in `getFactorValue` throws an exception when it is given the string ")".

19. We want to check whether any `queen[i]` attacks any `queen[j]`, but attacking is symmetric. That is, we can choose to compare only those for which $i < j$ (or, alternatively, those for which $i > j$). We don't want to call the `attacks` method when i equals j ; it would return true.

20. One solution:



21. Two solutions: The one from Self Check 20, and its mirror image.



WORKED EXAMPLE 13.1



Finding Files

Problem Statement Your task is to print the names of all files in a directory tree that end in a given extension.

To solve this task, you need to know two methods of the `File` class. A `File` object can represent either a directory or a regular file, and the method

```
boolean isDirectory()
```

tells you which it is. The method

```
File[] listFiles()
```

yields an array of all `File` objects describing the files and directories in a given directory. These can be directories or files.

For example, consider the directory tree at right, and suppose the `File` object `f` represents the `code` directory. Then `f.isDirectory()` returns true, and `f.listFiles()` returns an array containing `File` objects describing `code/ch01`, `code/ch02`, and `code/ch03`.

**Step 1**

Break the input into parts that can themselves be inputs to the problem.

Our problem has two inputs: A `File` object representing a directory tree, and an extension. Clearly, nothing is gained from manipulating the extension. However, there is an obvious way of chopping up the directory tree:

- Consider all files in the root level of the directory tree.
- Then consider each tree formed by a subdirectory.

This leads to a valid strategy. Find matching files in the root directory, and then recursively find them in each child subdirectory.

```

For each File object in the root
  If the File object is a directory
    Recursively find files in that directory.
  Else if the name ends in the extension
    Print the name.

```

Step 2

Combine solutions with simpler inputs into a solution of the original problem.

We are asked to simply print the files that we find, so there aren't any results to combine.

Had we been asked to produce an array list of the found files, we would place all matches of the root directory into an array list and add all results from the subdirectories into the same list.

Step 3 Find solutions to the simplest inputs.

The simplest input is a file that isn't a directory. In that case, we simply check whether it ends in the given extension, and if so, print it.

Step 4 Implement the solution by combining the simple cases and the reduction step.

We design a class `FileFinder` with a method for finding the matching files:

```
public class FileFinder
{
    private File[] children;

    /**
     * Constructs a file finder for a given directory tree.
     * @param startingDirectory the starting directory of the tree
     */
    public FileFinder(File startingDirectory)
    {
        children = startingDirectory.listFiles();
    }

    /**
     * Prints all files whose names end in a given extension.
     * @param extension a file extension (such as ".java")
     */
    public void find(String extension)
    {
        . . .
    }
}
```

In our case, the reduction step is simply to look at the files and subdirectories:

```
For each child in children
  If the child is a directory
    Recursively find files in the child.
  Else
    If the name of child ends in extension
      Print the name.
```

Here is the complete `FileFinder.find` method:

```
/**
 * Prints all files whose names end in a given extension.
 * @param extension a file extension (such as ".java")
 */
public void find(String extension)
{
    for (File child : children)
    {
        String fileName = child.toString();
        if (child.isDirectory())
        {
            FileFinder finder = new FileFinder(child);
            finder.find(extension);
        }
        else if (fileName.endsWith(extension))
        {
            System.out.println(fileName);
        }
    }
}
```

`FileFinderDemo.java` in the `worked_example_1` directory completes the solution.

In this solution, we used a class for each directory. Alternatively, we can use a recursive static method:

```
/*
    Prints all files whose names end in a given extension.
    @param aFile a file or directory
    @param extension a file extension (such as ".java")
*/
public static void find(File aFile, String extension)
{
    if (aFile.isDirectory())
    {
        for (File child : aFile.listFiles())
        {
            find(child, extension);
        }
    }
    else
    {
        String fileName = aFile.toString();
        if (fileName.endsWith(extension))
        {
            System.out.println(fileName);
        }
    }
}
```

The basic approach is the same. For a file, we check whether it ends in the given extension. If so, we print it. For a directory, we look at all the files and directories inside.

Here, we chose to accept either a file or directory. For that reason, the calling pattern is subtly different. In our first solution, recursive calls are only made on directories. In the second solution, the method is called recursively on all elements of the array returned by `listFiles()`. The recursion ends right away for files.

A complete solution is in the `ch13/worked_example_1` folder of your companion code.

worked_example_1/FileFinder2.java

```
1 import java.io.File;
2
3 public class FileFinder2
4 {
5     public static void main(String[] args)
6     {
7         File startingDirectory = new File("/home/myname");
8         find(startingDirectory, ".java");
9     }
10
11 /**
12  * Prints all files whose names end in a given extension.
13  * @param aFile a file or directory
14  * @param extension a file extension (such as ".java")
15  */
16 public static void find(File aFile, String extension)
17 {
18     if (aFile.isDirectory())
19     {
20         for (File child : aFile.listFiles())
21         {
22             find(child, extension);
23         }
24     }
25 }
```

WE4 Chapter 13 Recursion

```
23     }
24 }
25 else
26 {
27     String fileName = aFile.toString();
28     if (fileName.endsWith(extension))
29     {
30         System.out.println(fileName);
31     }
32 }
33 }
34 }
```



WORKED EXAMPLE 13.2

Towers of Hanoi



Problem Statement The “Towers of Hanoi” puzzle has a board with three pegs and a stack of disks of decreasing size, initially on the first peg (see Figure 7).

The goal is to move all disks to the third peg. One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

Legend has it that a temple (presumably in Hanoi) contains such an assembly, with sixty-four golden disks, which the priests move in the prescribed fashion. When they have arranged all disks on the third peg, the world will come to an end.

Let us help out by writing a program that prints instructions for moving the disks.

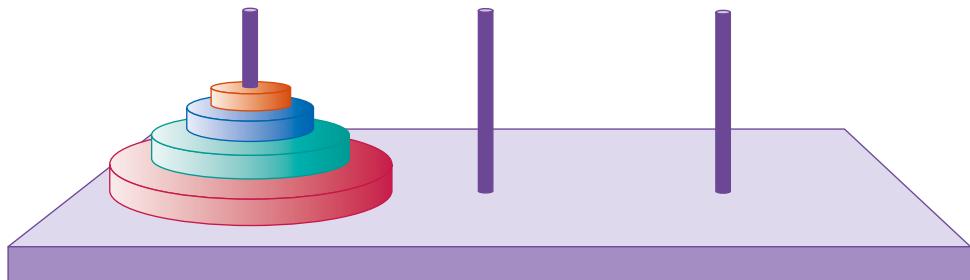


Figure 7 Towers of Hanoi

Consider the problem of moving d disks from peg p_1 to peg p_2 , where p_1 and p_2 are 1, 2, or 3, and $p_1 \neq p_2$. Since $1 + 2 + 3 = 6$, we can get the index of the remaining peg as $p_3 = 6 - p_1 - p_2$.

Now we can move the disks as follows:

- Move the top $d - 1$ disks from p_1 to p_3
- Move one disk (the one on the bottom of the pile of d disks) from p_1 to p_2
- Move the $d - 1$ disks that were parked on p_3 to p_2

The first and third step need to be handled recursively, but because we move one fewer disk, the recursion will eventually terminate.

It is very straightforward to translate the algorithm into Java. For the second step, we simply print out the instruction for the priest, something like

Move disk from peg 1 to 3

worked_example_2/TowersOfHanoiInstructions.java

```

1  /**
2   * This program prints instructions for solving a Towers of Hanoi puzzle.
3   */
4  public class TowersOfHanoiInstructions
5  {
6      public static void main(String[] args)
7      {
8          move(5, 1, 3);
9      }
10
11     /**
12      Print instructions for moving a pile of disks from one peg to another.
13      @param disks the number of disks to move
14      @param from the peg from which to move the disks
15      @param to the peg to which to move the disks

```

```

16  */
17  public static void move(int disks, int from, int to)
18  {
19      if (disks > 0)
20      {
21          int other = 6 - from - to;
22          move(disks - 1, from, other);
23          System.out.println("Move disk from peg " + from + " to " + to);
24          move(disks - 1, other, to);
25      }
26  }
27 }
```

Program Run

```

Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 3 to 2
Move disk from peg 3 to 1
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
Move disk from peg 1 to 2
Move disk from peg 3 to 2
Move disk from peg 1 to 3
Move disk from peg 2 to 1
Move disk from peg 2 to 3
Move disk from peg 1 to 3
```

These instructions may suffice for the priests, but unfortunately it is not easy for us to see what is going on. Let's improve the program so that it actually carries out the instructions and shows the contents of the towers after each move.

We use a class `Tower` that manages the disks in one tower. Each disk is represented as an integer indicating its size from 1 to n , the number of disks in the puzzle.

We provide methods to remove the top disk, to add a disk to the top, and to show the contents of the tower as a list of disk sizes, for example, [5, 4, 1].

worked_example_2/Tower.java

```

1 import java.util.ArrayList;
2
3 /**
4  * A tower containing disks in the Towers of Hanoi puzzle.
5 */
6 public class Tower
7 {
8     private ArrayList<Integer> disks;
9
10    /**
11     * Constructs a tower holding a given number of disks of decreasing size.
12     * @param ndisks the number of disks
13    */
14    public Tower(int ndisks)
15    {
16        disks = new ArrayList<Integer>();
17        for (int d = ndisks; d >= 1; d--) { disks.add(d); }
18    }
19
20    /**
21     * Removes the top disk from this tower.
22     * @return the size of the removed disk
23    */
24    public int remove()
25    {
26        return disks.remove(disks.size() - 1);
27    }
28
29    /**
30     * Adds a disk to this tower.
31     * @param size the size of the disk to add
32    */
33    public void add(int size)
34    {
35        if (disks.size() > 0 && disks.get(disks.size() - 1) < size)
36        {
37            throw new IllegalStateException("Disk is too large");
38        }
39        disks.add(size);
40    }
41
42    public String toString() { return disks.toString(); }
43}

```

A TowersOfHanoi puzzle has three towers:

```

public class TowersOfHanoi
{
    private Tower[] towers;

    public TowersOfHanoi(int ndisks)
    {
        towers = new Tower[3];
        towers[0] = new Tower(ndisks);
        towers[1] = new Tower(0);
        towers[2] = new Tower(0);
    }
    . .

```

WE8 Chapter 13 Recursion

```
}
```

Its move method first carries out the move, then prints the contents of the towers:

```
public void move(int disks, int from, int to)
{
    if (disks > 0)
    {
        int other = 3 - from - to;
        move(disks - 1, from, other);
        towers[to].add(towers[from].remove());
        System.out.println(Arrays.toString(towers));
        move(disks - 1, other, to);
    }
}
```

Here, we changed the index values to 0, 1, 2. Therefore, the index of the other peg is $3 - \text{from} - \text{to}$.

Here is the main method:

```
public static void main(String[] args)
{
    final int NDISKS = 5;
    TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
    towers.move(NDISKS, 0, 2);
}
```

The program output is

```
[[5, 4, 3, 2], [], [1]]
[[5, 4, 3], [2], [1]]
[[5, 4, 3], [2, 1], []]
[[5, 4], [2, 1], [3]]
[[5, 4, 1], [2], [3]]
[[5, 4, 1], [], [3, 2]]
[[5, 4], [], [3, 2, 1]]
[[5], [4], [3, 2, 1]]
[[5], [4, 1], [3, 2]]
[[5, 2], [4, 1], [3]]
[[5, 2, 1], [4], [3]]
[[5, 2, 1], [4, 3], []]
[[5, 2], [4, 3], [1]]
[[5], [4, 3, 2], [1]]
[[5], [4, 3, 2, 1], []]
[[], [4, 3, 2, 1], [5]]
[[1], [4, 3, 2], [5]]
[[1], [4, 3], [5, 2]]
[[], [4, 3], [5, 2, 1]]
[[3], [4], [5, 2, 1]]
[[3], [4, 1], [5, 2]]
[[3, 2], [4, 1], [5]]
[[3, 2, 1], [4], [5]]
[[3, 2, 1], [], [5, 4]]
[[3, 2], [], [5, 4, 1]]
[[3], [2], [5, 4, 1]]
[[3], [2, 1], [5, 4]]
[[], [2, 1], [5, 4, 3]]
[[1], [2], [5, 4, 3]]
[[1], [], [5, 4, 3, 2]]
[[], [], [5, 4, 3, 2, 1]]
```

That's better. Now you can see how the disks move. You can check that all moves are legal—the disk size always decreases.

You can see that it takes $31 = 2^5 - 1$ moves to solve the puzzle for 5 disks. With 64 disks, it takes $2^{64} - 1 = 18446744073709551615$ moves. If the priests can move one disk per second, it takes about 585 billion years to finish the job. Because the earth is about 4.5 billion years old at the time this book is written, we don't have to worry too much whether the world will really come to an end when they are done.

worked_example_2/TowersOfHanoiDemo.java

```

1 import java.util.ArrayList;
2 import java.util.Arrays;
3
4 /**
5     This program shows a solution for a Towers of Hanoi puzzle.
6 */
7 public class TowersOfHanoiDemo
8 {
9     public static void main(String[] args)
10    {
11        final int NDISKS = 5;
12        TowersOfHanoi towers = new TowersOfHanoi(NDISKS);
13        towers.move(NDISKS, 0, 2);
14    }
15 }
```

worked_example_2/TowersOfHanoi.java

```

1 import java.util.Arrays;
2
3 /**
4     A Towers of Hanoi puzzle with three towers.
5 */
6 public class TowersOfHanoi
7 {
8     private Tower[] towers;
9
10    /**
11     Constructs a puzzle in which the first tower has a given number of disks.
12     @param ndisks the number of disks
13    */
14    public TowersOfHanoi(int ndisks)
15    {
16        towers = new Tower[3];
17        towers[0] = new Tower(ndisks);
18        towers[1] = new Tower(0);
19        towers[2] = new Tower(0);
20    }
21
22    /**
23     Moves a pile of disks from one peg to another and displays the movement.
24     @param disks the number of disks to move
25     @param from the peg from which to move the disks
26     @param to the peg to which to move the disks
27    */
28    public void move(int disks, int from, int to)
29    {
30        if (disks > 0)
31        {
32            int other = 3 - from - to;
33            move(disks - 1, from, other);
34            move(1, from, to);
35            move(disks - 1, other, to);
36        }
37    }
38 }
```

```
34     towers[to].add(towers[from].remove());
35     System.out.println(Arrays.toString(towers));
36     move(disks - 1, other, to);
37   }
38 }
39 }
```

SORTING AND SEARCHING

CHAPTER GOALS

- To study several sorting and searching algorithms
- To appreciate that algorithms for the same task can differ widely in performance
- To understand the big-Oh notation
- To estimate and compare the performance of algorithms
- To write code to measure the running time of a program



© Volkan Ersoy/iStockphoto.

CHAPTER CONTENTS

14.1 SELECTION SORT	650
14.2 PROFILING THE SELECTION SORT ALGORITHM	653
14.3 ANALYZING THE PERFORMANCE OF THE SELECTION SORT ALGORITHM	656
ST1 Oh, Omega, and Theta	658
ST2 Insertion Sort	659
14.4 MERGE SORT	661
14.5 ANALYZING THE MERGE SORT ALGORITHM	664
ST3 The Quicksort Algorithm	666
14.6 SEARCHING	668
C&S The First Programmer	672

14.7 PROBLEM SOLVING: ESTIMATING THE RUNNING TIME OF AN ALGORITHM 673

14.8 SORTING AND SEARCHING IN THE JAVA LIBRARY 678

- CE1** The compareTo Method Can Return Any Integer, Not Just -1, 0, and 1 680
- ST4** The Comparator Interface 680
- J81** Comparators with Lambda Expressions 681
- WE1** Enhancing the Insertion Sort Algorithm 



© Volkan Ersoy/iStockphoto.

One of the most common tasks in data processing is sorting. For example, an array of employees often needs to be displayed in alphabetical order or sorted by salary. In this chapter, you will learn several sorting methods as well as techniques for comparing their performance. These techniques are useful not just for sorting algorithms, but also for analyzing other algorithms.

Once an array of elements is sorted, one can rapidly locate individual elements. You will study the *binary search* algorithm that carries out this fast lookup.

14.1 Selection Sort

In this section, we show you the first of several sorting algorithms. A *sorting algorithm* rearranges the elements of a collection so that they are stored in sorted order. To keep the examples simple, we will discuss how to sort an array of integers before going on to sorting strings or more complex data. Consider the following array a:

[0]	[1]	[2]	[3]	[4]
11	9	17	5	12

The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

An obvious first step is to find the smallest element. In this case the smallest element is 5, stored in $a[3]$. We should move the 5 to the beginning of the array. Of course, there is already an element stored in $a[0]$, namely 11. Therefore we cannot simply move $a[3]$ into $a[0]$ without moving the 11 somewhere else. We don't yet know where the 11 should end up, but we know for certain that it should not be in $a[0]$. We simply get it out of the way by *swapping* it with $a[3]$:

[0]	[1]	[2]	[3]	[4]
5	9	17	11	12
			↑	↑

Now the first element is in the correct place. The darker color in the figure indicates the portion of the array that is already sorted.

In selection sort, pick the smallest element and swap it with the first one. Pick the smallest element of the remaining ones and swap it with the next one, and so on.



© Zone Creative/iStockphoto.

Next we take the minimum of the remaining entries $a[1] \dots a[4]$. That minimum value, 9, is already in the correct place. We don't need to do anything in this case and can simply extend the sorted area by one to the right:

[0]	[1]	[2]	[3]	[4]
5	9	17	11	12

Repeat the process. The minimum value of the unsorted region is 11, which needs to be swapped with the first value of the unsorted region, 17:

[0]	[1]	[2]	[3]	[4]
5	9	11	17	12

Now the unsorted region is only two elements long, but we keep to the same successful strategy. The minimum value is 12, and we swap it with the first value, 17:

[0]	[1]	[2]	[3]	[4]
5	9	11	12	17

That leaves us with an unprocessed region of length 1, but of course a region of length 1 is always sorted. We are done.

Let's program this algorithm, called **selection sort**. For this program, as well as the other programs in this chapter, we will use a utility method to generate an array with random entries. We place it into a class `ArrayUtil` so that we don't have to repeat the code in every example. To show the array, we call the static `toString` method of the `Arrays` class in the Java library and print the resulting string (see Section 6.3.4). We also add a method for swapping elements to the `ArrayUtil` class. (See Section 6.3.8 for details about swapping array elements.)

This algorithm will sort any array of integers. If speed were not an issue, or if there were no better sorting method available, we could stop the discussion of sorting right here. As the next section shows, however, this algorithm, while entirely correct, shows disappointing performance when run on a large data set.

Special Topic 14.2 discusses insertion sort, another simple sorting algorithm.

sec01/SelectionSorter.java

```

1  /**
2   * The sort method of this class sorts an array, using the selection
3   * sort algorithm.
4  */
5  public class SelectionSorter
6  {
7      /**
8       * Sorts an array, using selection sort.
9       * @param a the array to sort
10    */
11   public static void sort(int[] a)
12   {
13       for (int i = 0; i < a.length - 1; i++)
14       {
15           int minPos = minimumPosition(a, i);
16           ArrayUtil.swap(a, minPos, i);
17       }
18   }
}

```

```

19
20  /**
21   * Finds the smallest element in a tail range of the array.
22   * @param a the array to sort
23   * @param from the first position in a to compare
24   * @return the position of the smallest element in the
25   * range a[from] . . . a[a.length - 1]
26  */
27  private static int minimumPosition(int[] a, int from)
28  {
29      int minPos = from;
30      for (int i = from + 1; i < a.length; i++)
31      {
32          if (a[i] < a[minPos]) { minPos = i; }
33      }
34      return minPos;
35  }
36 }

```

sec01/SelectionSortDemo.java

```

1 import java.util.Arrays;
2
3 /**
4  * This program demonstrates the selection sort algorithm by
5  * sorting an array that is filled with random numbers.
6  */
7 public class SelectionSortDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        SelectionSorter.sort(a);
15
16        System.out.println(Arrays.toString(a));
17    }
18 }

```

sec01/ArrayUtil.java

```

1 import java.util.Random;
2
3 /**
4  * This class contains utility methods for array manipulation.
5  */
6 public class ArrayUtil
7 {
8     private static Random generator = new Random();
9
10    /**
11     * Creates an array filled with random values.
12     * @param length the length of the array
13     * @param n the number of possible random values
14     * @return an array filled with length numbers between
15     * 0 and n - 1
16    */
17    public static int[] randomIntArray(int length, int n)
18    {

```

```

19     int[] a = new int[length];
20     for (int i = 0; i < a.length; i++)
21     {
22         a[i] = generator.nextInt(n);
23     }
24
25     return a;
26 }
27
28 /**
29  * Swaps two entries of an array.
30  * @param a the array
31  * @param i the first position to swap
32  * @param j the second position to swap
33 */
34 public static void swap(int[] a, int i, int j)
35 {
36     int temp = a[i];
37     a[i] = a[j];
38     a[j] = temp;
39 }
40 }
```

Program Run

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77, 73, 87, 36, 81]
[2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52, 65, 73, 77, 81, 87, 89, 96, 99]
```

SELF CHECK



1. Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?
2. What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?
3. How can you change the selection sort algorithm so that it sorts the elements in descending order (that is, with the largest element at the beginning of the array)?
4. Suppose we modified the selection sort algorithm to start at the end of the array, working toward the beginning. In each step, the current position is swapped with the minimum. What is the result of this modification?

Practice It

Now you can try these exercises at the end of the chapter: R14.2, R14.12, E14.1, E14.2.

14.2 Profiling the Selection Sort Algorithm

To measure the performance of a program, you could simply run it and use a stopwatch to measure how long it takes. However, most of our programs run very quickly, and it is not easy to time them accurately in this way. Furthermore, when a program takes a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory and displaying the result (for which we should not penalize it).

In order to measure the running time of an algorithm more accurately, we will create a `Stopwatch` class. This class works like a real stopwatch. You can start it, stop

it, and read out the elapsed time. The class uses the `System.currentTimeMillis` method, which returns the milliseconds that have elapsed since midnight at the start of January 1, 1970. Of course, you don't care about the absolute number of seconds since this historical moment, but the *difference* of two such counts gives us the number of milliseconds in a given time interval.

Here is the code for the `StopWatch` class:

sec02/StopWatch.java

```

1  /**
2   * A stopwatch accumulates time when it is running. You can
3   * repeatedly start and stop the stopwatch. You can use a
4   * stopwatch to measure the running time of a program.
5  */
6  public class StopWatch
7  {
8      private long elapsedTime;
9      private long startTime;
10     private boolean isRunning;
11
12     /**
13      Constructs a stopwatch that is in the stopped state
14      and has no time accumulated.
15     */
16     public StopWatch()
17     {
18         reset();
19     }
20
21     /**
22      Starts the stopwatch. Time starts accumulating now.
23     */
24     public void start()
25     {
26         if (isRunning) { return; }
27         isRunning = true;
28         startTime = System.currentTimeMillis();
29     }
30
31     /**
32      Stops the stopwatch. Time stops accumulating and is
33      is added to the elapsed time.
34     */
35     public void stop()
36     {
37         if (!isRunning) { return; }
38         isRunning = false;
39         long endTime = System.currentTimeMillis();
40         elapsedTime = elapsedTime + endTime - startTime;
41     }
42
43     /**
44      Returns the total elapsed time.
45      @return the total elapsed time
46     */
47     public long getElapsedTime()
48     {
49         if (isRunning)
50         {

```

```

51     long endTime = System.currentTimeMillis();
52     return elapsedTime + endTime - startTime;
53   }
54   else
55   {
56     return elapsedTime;
57   }
58 }
59
60 /**
61  * Stops the watch and resets the elapsed time to 0.
62  */
63 public void reset()
64 {
65   elapsedTime = 0;
66   isRunning = false;
67 }
68 }
```

Here is how to use the stopwatch to measure the sorting algorithm's performance:

sec02/SelectionSortTimer.java

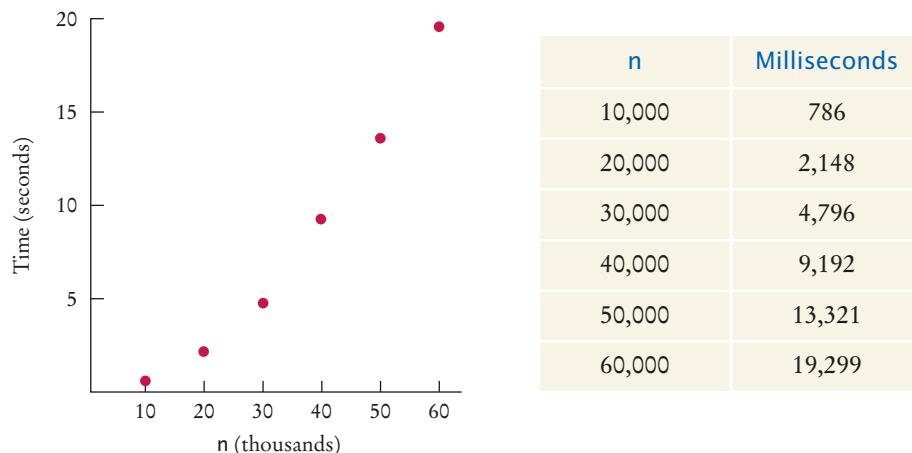
```

1 import java.util.Scanner;
2
3 /**
4  * This program measures how long it takes to sort an
5  * array of a user-specified size with the selection
6  * sort algorithm.
7 */
8 public class SelectionSortTimer
9 {
10   public static void main(String[] args)
11   {
12     Scanner in = new Scanner(System.in);
13     System.out.print("Enter array size: ");
14     int n = in.nextInt();
15
16     // Construct random array
17
18     int[] a = ArrayUtil.randomIntArray(n, 100);
19
20     // Use stopwatch to time selection sort
21
22     StopWatch timer = new StopWatch();
23
24     timer.start();
25     SelectionSorter.sort(a);
26     timer.stop();
27
28     System.out.println("Elapsed time: "
29                     + timer.getElapsedTime() + " milliseconds");
30   }
31 }
```

Program Run

```

Enter array size: 50000
Elapsed time: 13321 milliseconds
```

**Figure 1** Time Taken by Selection Sort

To measure the running time of a method, get the current time immediately before and after the method call.

By starting to measure the time just before sorting, and stopping the stopwatch just after, you get the time required for the sorting process, without counting the time for input and output.

The table in Figure 1 shows the results of some sample runs. These measurements were obtained with an Intel processor with a clock speed of 2 GHz, running Java 6 on the Linux operating system. On another computer the actual numbers will look different, but the relationship between the numbers will be the same.

The graph in Figure 1 shows a plot of the measurements. As you can see, when you double the size of the data set, it takes about four times as long to sort it.



5. Approximately how many seconds would it take to sort a data set of 80,000 values?
6. Look at the graph in Figure 1. What mathematical shape does it resemble?

Practice It Now you can try these exercises at the end of the chapter: E14.3, E14.9.

14.3 Analyzing the Performance of the Selection Sort Algorithm

Let us count the number of operations that the program must carry out to sort an array with the selection sort algorithm. We don't actually know how many machine operations are generated for each Java instruction, or which of those instructions are more time-consuming than others, but we can make a simplification. We will simply count how often an array element is *visited*. Each visit requires about the same amount of work by other operations, such as incrementing subscripts and comparing values.

Let n be the size of the array. First, we must find the smallest of n numbers. To achieve that, we must visit n array elements. Then we swap the elements, which takes

two visits. (You may argue that there is a certain probability that we don't need to swap the values. That is true, and one can refine the computation to reflect that observation. As we will soon see, doing so would not affect the overall conclusion.) In the next step, we need to visit only $n - 1$ elements to find the minimum. In the following step, $n - 2$ elements are visited to find the minimum. The last step visits two elements to find the minimum. Each step requires two visits to swap the elements. Therefore, the total number of visits is

$$\begin{aligned} n + 2 + (n - 1) + 2 + \cdots + 2 + 2 &= (n + (n - 1) + \cdots + 2) + (n - 1) \cdot 2 \\ &= (2 + \cdots + (n - 1) + n) + (n - 1) \cdot 2 \\ &= \frac{n(n + 1)}{2} - 1 + (n - 1) \cdot 2 \end{aligned}$$

because

$$1 + 2 + \cdots + (n - 1) + n = \frac{n(n + 1)}{2}$$

After multiplying out and collecting terms of n , we find that the number of visits is

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We obtain a quadratic equation in n . That explains why the graph of Figure 1 looks approximately like a parabola.

Now simplify the analysis further. When you plug in a large value for n (for example, 1,000 or 2,000), then $\frac{1}{2}n^2$ is 500,000 or 2,000,000. The lower term, $\frac{5}{2}n - 3$, doesn't contribute much at all; it is only 2,497 or 4,997, a drop in the bucket compared to the hundreds of thousands or even millions of comparisons specified by the $\frac{1}{2}n^2$ term. We will just ignore these lower-level terms. Next, we will ignore the constant factor $\frac{1}{2}$. We are not interested in the actual count of visits for a single n . We want to compare the ratios of counts for different values of n . For example, we can say that sorting an array of 2,000 numbers requires four times as many visits as sorting an array of 1,000 numbers:

$$\frac{\left(\frac{1}{2} \cdot 2000^2\right)}{\left(\frac{1}{2} \cdot 1000^2\right)} = 4$$

The factor $\frac{1}{2}$ cancels out in comparisons of this kind. We will simply say, "The number of visits is of order n^2 ." That way, we can easily see that the number of comparisons increases fourfold when the size of the array doubles: $(2n)^2 = 4n^2$.

To indicate that the number of visits is of order n^2 , computer scientists often use **big-Oh notation**: The number of visits is $O(n^2)$. This is a convenient shorthand. (See Special Topic 14.1 for a formal definition.)

To turn a polynomial expression such as

$$\frac{1}{2}n^2 + \frac{5}{2}n - 3$$

into big-Oh notation, simply locate the fastest-growing term, n^2 , and ignore its constant coefficient, no matter how large or small it may be.

We observed before that the actual number of machine operations, and the actual amount of time that the computer spends on them, is approximately proportional to the number of element visits. Maybe there are about 10 machine operations

Computer scientists use the big-Oh notation to describe the growth rate of a function.

Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.

(increments, comparisons, memory loads, and stores) for every element visit. The number of machine operations is then approximately $10 \times \frac{1}{2}n^2$. As before, we aren't interested in the coefficient, so we can say that the number of machine operations, and hence the time spent on the sorting, is of the order n^2 or $O(n^2)$.

The sad fact remains that doubling the size of the array causes a fourfold increase in the time required for sorting it with selection sort. When the size of the array increases by a factor of 100, the sorting time increases by a factor of 10,000. To sort an array of a million entries (for example, to create a telephone directory), takes 10,000 times as long as sorting 10,000 entries. If 10,000 entries can be sorted in about 3/4 of a second (as in our example), then sorting one million entries requires well over two hours. We will see in the next section how one can dramatically improve the performance of the sorting process by choosing a more sophisticated algorithm.

SELF CHECK



7. If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?
8. How large does n need to be so that $\frac{1}{2}n^2$ is bigger than $\frac{5}{2}n - 3$?
9. Section 6.3.6 has two algorithms for removing an element from an array of length n . How many array visits does each algorithm require on average?
10. Describe the number of array visits in Self Check 9 using the big-Oh notation.
11. What is the big-Oh running time of checking whether an array is already sorted?
12. Consider this algorithm for sorting an array. Set k to the length of the array. Find the maximum of the first k elements. Remove it, using the second algorithm of Section 6.3.6. Decrement k and place the removed element into the k th position. Stop if k is 1. What is the algorithm's running time in big-Oh notation?

Practice It Now you can try these exercises at the end of the chapter: R14.4, R14.6, R14.8.

Special Topic 14.1



Oh, Omega, and Theta

We have used the big-Oh notation somewhat casually in this chapter to describe the growth behavior of a function. Here is the formal definition of the big-Oh notation: Suppose we have a function $T(n)$. Usually, it represents the processing time of an algorithm for a given input of size n . But it could be any function. Also, suppose that we have another function $f(n)$. It is usually chosen to be a simple function, such as $f(n) = n^k$ or $f(n) = \log(n)$, but it too can be any function. We write

$$T(n) = O(f(n))$$

if $T(n)$ grows at a rate that is bounded by $f(n)$. More formally, we require that for all n larger than some threshold, the ratio $T(n)/f(n) \leq C$ for some constant value C .

If $T(n)$ is a polynomial of degree k in n , then one can show that $T(n) = O(n^k)$. Later in this chapter, we will encounter functions that are $O(\log(n))$ or $O(n \log(n))$. Some algorithms take much more time. For example, one way of sorting a sequence is to compute all of its permutations, until you find one that is in increasing order. Such an algorithm takes $O(n!)$ time, which is very bad indeed.

Table 1 shows common big-Oh expressions, sorted by increasing growth.

Strictly speaking, $T(n) = O(f(n))$ means that T grows no faster than f . But it is permissible for T to grow much more slowly. Thus, it is technically correct to state that $T(n) = n^2 + 5n - 3$ is $O(n^3)$ or even $O(n^{10})$.

Table 1 Common Big-Oh Growth Rates

Big-Oh Expression	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(n)$	Linear
$O(n \log(n))$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

Computer scientists have invented additional notation to describe the growth behavior of functions more accurately. The expression

$$T(n) = \Omega(f(n))$$

means that T grows at least as fast as f , or, formally, that for all n larger than some threshold, the ratio $T(n)/f(n) \geq C$ for some constant value C . (The Ω symbol is the capital Greek letter omega.) For example, $T(n) = n^2 + 5n - 3$ is $\Omega(n^2)$ or even $\Omega(n)$.

The expression

$$T(n) = \Theta(f(n))$$

means that T and f grow at the same rate—that is, both $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$ hold. (The Θ symbol is the capital Greek letter theta.)

The Θ notation gives the most precise description of growth behavior. For example, $T(n) = n^2 + 5n - 3$ is $\Theta(n^2)$ but not $\Theta(n)$ or $\Theta(n^3)$.

The notations are very important for the precise analysis of algorithms. However, in casual conversation it is common to stick with big-Oh, while still giving an estimate as good as one can make.

Special Topic 14.2



Insertion Sort

Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence

$a[0] \ a[1] \ \dots \ a[k]$

of an array is already sorted. (When the algorithm starts, we set k to 0.) We enlarge the initial sequence by inserting the next array element, $a[k + 1]$, at the proper location. When we reach the end of the array, the sorting process is complete.

For example, suppose we start with the array

11 9 16 5 7

Of course, the initial sequence of length 1 is already sorted. We now add $a[1]$, which has the value 9. The element needs to be inserted before the element 11. The result is

9 11 16 5 7

Next, we add $a[2]$, which has the value 16. This element does not have to be moved.

9	11	16	5	7
---	----	----	---	---

We repeat the process, inserting $a[3]$ or 5 at the very beginning of the initial sequence.

5	9	11	16	7
---	---	----	----	---

Finally, $a[4]$ or 7 is inserted in its correct position, and the sorting is completed.

The following class implements the insertion sort algorithm:

```
public class InsertionSorter
{
    /**
     * Sorts an array, using insertion sort.
     * @param a the array to sort
     */
    public static void sort(int[] a)
    {
        for (int i = 1; i < a.length; i++)
        {
            int next = a[i];
            // Move all larger elements up
            int j = i;
            while (j > 0 && a[j - 1] > next)
            {
                a[j] = a[j - 1];
                j--;
            }
            // Insert the element
            a[j] = next;
        }
    }
}
```

How efficient is this algorithm? Let n denote the size of the array. We carry out $n - 1$ iterations. In the k th iteration, we have a sequence of k elements that is already sorted, and we need to insert a new element into the sequence. For each insertion, we need to visit the elements of the initial sequence until we have found the location in which the new element can be inserted. Then we need to move up the remaining elements of the sequence. Thus, $k + 1$ array elements are visited. Therefore, the total number of visits is

$$2 + 3 + \dots + n = \frac{n(n+1)}{2} - 1$$

We conclude that insertion sort is an $O(n^2)$ algorithm, on the same order of efficiency as selection sort.

Insertion sort has a desirable property: Its performance is $O(n)$ if the array is already sorted—see Exercise R14.19. This is a useful property in practical applications, in which data sets are often partially sorted.

Insertion sort is an $O(n^2)$ algorithm.



FULL CODE EXAMPLE
Go to wiley.com/go/bj102code to download a program that illustrates sorting with insertion sort.

Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.



© Kirby Hamilton/Stockphoto.

14.4 Merge Sort

In this section, you will learn about the **merge sort** algorithm, a much more efficient algorithm than selection sort. The basic idea behind merge sort is very simple.

Suppose we have an array of 10 integers. Let us engage in a bit of wishful thinking and hope that the first half of the array is already perfectly sorted, and the second half is too, like this:

5	9	10	12	17	1	8	11	20	32
---	---	----	----	----	---	---	----	----	----

Now it is simple to *merge* the two sorted arrays into one sorted array, by taking a new element from either the first or the second subarray, and choosing the smaller of the elements each time:

5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32
5	9	10	12	17	1	8	11	20	32

1									
1	5								
1	5	8							
1	5	8	9						
1	5	8	9	10					
1	5	8	9	10	11				
1	5	8	9	10	11	12			
1	5	8	9	10	11	12	17		
1	5	8	9	10	11	12	17	20	
1	5	8	9	10	11	12	17	20	32

In fact, you may have performed this merging before if you and a friend had to sort a pile of papers. You and the friend split the pile in half, each of you sorted your half, and then you merged the results together.

That is all well and good, but it doesn't seem to solve the problem for the computer. It still must sort the first and second halves of the array, because it can't very well ask a few buddies to pitch in. As it turns out, though, if the computer keeps dividing the array into smaller and smaller subarrays, sorting each half and merging them back together, it carries out dramatically fewer steps than the selection sort requires.

Let's write a `MergeSorter` class that implements this idea. When the `MergeSorter` sorts an array, it makes two arrays, each half the size of the original, and sorts them recursively. Then it merges the two sorted arrays together:

```
public static void sort(int[] a)
{
    if (a.length <= 1) { return; }
    int[] first = new int[a.length / 2];
    int[] second = new int[a.length - first.length];
    // Copy the first half of a into first, the second half into second
    .
    .
    sort(first);
    sort(second);
    merge(first, second, a);
}
```



© Rich Legg/Stockphoto.

In merge sort, one sorts each half, then merges the sorted halves.

The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

The `merge` method is tedious but quite straightforward. You will find it in the code that follows.

sec04/MergeSorter.java

```

1  /**
2   * The sort method of this class sorts an array, using the merge
3   * sort algorithm.
4  */
5  public class MergeSorter
6  {
7      /**
8       * Sorts an array, using merge sort.
9       * @param a the array to sort
10    */
11   public static void sort(int[] a)
12   {
13       if (a.length <= 1) { return; }
14       int[] first = new int[a.length / 2];
15       int[] second = new int[a.length - first.length];
16       // Copy the first half of a into first, the second half into second
17       for (int i = 0; i < first.length; i++)
18       {
19           first[i] = a[i];
20       }
21       for (int i = 0; i < second.length; i++)
22       {
23           second[i] = a[first.length + i];
24       }
25       sort(first);
26       sort(second);
27       merge(first, second, a);
28   }
29
30 /**
31  * Merges two sorted arrays into an array.
32  * @param first the first sorted array
33  * @param second the second sorted array
34  * @param a the array into which to merge first and second
35  */
36 private static void merge(int[] first, int[] second, int[] a)
37 {
38     int iFirst = 0; // Next element to consider in the first array
39     int iSecond = 0; // Next element to consider in the second array
40     int j = 0; // Next open position in a
41
42     // As long as neither iFirst nor iSecond past the end, move
43     // the smaller element into a
44     while (iFirst < first.length && iSecond < second.length)
45     {
46         if (first[iFirst] < second[iSecond])
47         {
48             a[j] = first[iFirst];
49             iFirst++;
50         }
51         else
52         {
53             a[j] = second[iSecond];
54             iSecond++;
55         }
56     }
57 }
```

```

55     }
56     j++;
57 }
58
59 // Note that only one of the two loops below copies entries
60 // Copy any remaining entries of the first array
61 while (iFirst < first.length)
62 {
63     a[j] = first[iFirst];
64     iFirst++; j++;
65 }
66 // Copy any remaining entries of the second half
67 while (iSecond < second.length)
68 {
69     a[j] = second[iSecond];
70     iSecond++; j++;
71 }
72 }
73 }

```

sec04/MergeSortDemo.java

```

1 import java.util.Arrays;
2
3 /**
4  * This program demonstrates the merge sort algorithm by
5  * sorting an array that is filled with random numbers.
6  */
7 public class MergeSortDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13
14        MergeSorter.sort(a);
15
16        System.out.println(Arrays.toString(a));
17    }
18 }

```

Program Run

```
[8, 81, 48, 53, 46, 70, 98, 42, 27, 76, 33, 24, 2, 76, 62, 89, 90, 5, 13, 21]
[2, 5, 8, 13, 21, 24, 27, 33, 42, 46, 48, 53, 62, 70, 76, 76, 81, 89, 90, 98]
```

SELF CHECK

13. Why does only one of the two `while` loops at the end of the `merge` method do any work?
14. Manually run the merge sort algorithm on the array 8 7 6 5 4 3 2 1.
15. The merge sort algorithm processes an array by recursively processing two halves. Describe a similar recursive algorithm for computing the sum of all elements in an array.

Practice It Now you can try these exercises at the end of the chapter: R14.13, E14.4, E14.14.

14.5 Analyzing the Merge Sort Algorithm

The merge sort algorithm looks a lot more complicated than the selection sort algorithm, and it appears that it may well take much longer to carry out these repeated subdivisions. However, the timing results for merge sort look much better than those for selection sort.

Figure 2 shows a table and a graph comparing both sets of performance data. As you can see, merge sort is a tremendous improvement. To understand why, let us estimate the number of array element visits that are required to sort an array with the merge sort algorithm. First, let us tackle the merge process that happens after the first and second halves have been sorted.

Each step in the merge process adds one more element to a . That element may come from first or second, and in most cases the elements from the two halves must be compared to see which one to take. We'll count that as 3 visits (one for a and one each for first and second) per element, or $3n$ visits total, where n denotes the length of a . Moreover, at the beginning, we had to copy from a to first and second, yielding another $2n$ visits, for a total of $5n$.

If we let $T(n)$ denote the number of visits required to sort a range of n elements through the merge sort process, then we obtain

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 5n$$

because sorting each half takes $T(n/2)$ visits. Actually, if n is not even, then we have one subarray of size $(n - 1)/2$ and one of size $(n + 1)/2$. Although it turns out that this detail does not affect the outcome of the computation, we will nevertheless assume for now that n is a power of 2, say $n = 2^m$. That way, all subarrays can be evenly divided into two parts.

Unfortunately, the formula

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n$$

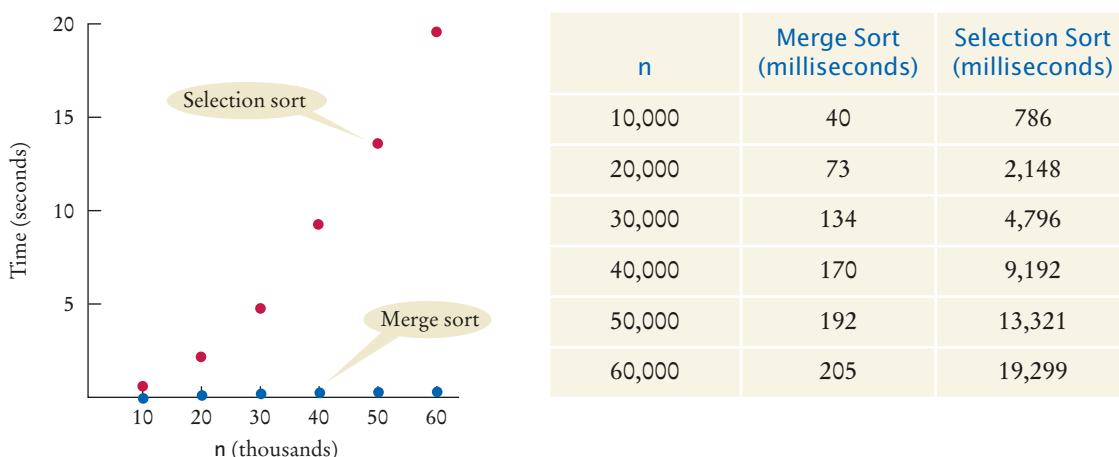


Figure 2 Time Taken by Merge Sort and Selection Sort

does not clearly tell us the relationship between n and $T(n)$. To understand the relationship, let us evaluate $T(n/2)$, using the same formula:

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 5\frac{n}{2}$$

Therefore

$$T(n) = 2 \times 2T\left(\frac{n}{4}\right) + 5n + 5n$$

Let us do that again:

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 5\frac{n}{4}$$

hence

$$T(n) = 2 \times 2 \times 2T\left(\frac{n}{8}\right) + 5n + 5n + 5n$$

This generalizes from 2, 4, 8, to arbitrary powers of 2:

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 5nk$$

Recall that we assume that $n = 2^m$; hence, for $k = m$,

$$\begin{aligned} T(n) &= 2^m T\left(\frac{n}{2^m}\right) + 5nm \\ &= nT(1) + 5nm \\ &= n + 5n \log_2(n) \end{aligned}$$

Because $n = 2^m$, we have $m = \log_2(n)$.

To establish the growth order, we drop the lower-order term n and are left with $5n \log_2(n)$. We drop the constant factor 5. It is also customary to drop the base of the logarithm, because all logarithms are related by a constant factor. For example,

$$\log_2(x) = \log_{10}(x)/\log_{10}(2) \approx \log_{10}(x) \times 3.32193$$

Hence we say that merge sort is an $O(n \log(n))$ algorithm.

Is the $O(n \log(n))$ merge sort algorithm better than the $O(n^2)$ selection sort algorithm? You bet it is. Recall that it took $100^2 = 10,000$ times as long to sort a million records as it took to sort 10,000 records with the $O(n^2)$ algorithm. With the $O(n \log(n))$ algorithm, the ratio is

$$\frac{1,000,000 \log(1,000,000)}{10,000 \log(10,000)} = 100 \left(\frac{6}{4} \right) = 150$$

Suppose for the moment that merge sort takes the same time as selection sort to sort an array of 10,000 integers, that is, $3/4$ of a second on the test machine. (Actually, it is much faster than that.) Then it would take about 0.75×150 seconds, or under two minutes, to sort a million integers. Contrast that with selection sort, which would take over two hours for the same task. As you can see, even if it takes you several hours to learn about a better algorithm, that can be time well spent.

Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2 .

**FULL CODE EXAMPLE**

Go to wiley.com/go/bj102code to download a program for timing the merge sort algorithm.

In this chapter we have barely begun to scratch the surface of this interesting topic. There are many sorting algorithms, some with even better performance than merge sort, and the analysis of these algorithms can be quite challenging. These important issues are often revisited in later computer science courses.

SELF CHECK

16. Given the timing data for the merge sort algorithm in the table at the beginning of this section, how long would it take to sort an array of 100,000 values?
17. If you double the size of an array, how much longer will the merge sort algorithm take to sort the new array?

Practice It Now you can try these exercises at the end of the chapter: R14.7, R14.16, R14.18.

Special Topic 14.3**The Quicksort Algorithm**

Quicksort is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a range $a[from] \dots a[to]$ of the array a , first rearrange the elements in the range so that no element in the range $a[from] \dots a[p]$ is larger than any element in the range $a[p + 1] \dots a[to]$. This step is called *partitioning* the range.

For example, suppose we start with a range

5	3	2	6	4	1	3	7
---	---	---	---	---	---	---	---

Here is a partitioning of the range. Note that the partitions aren't yet sorted.

3	3	2	1	4			6	5	7
---	---	---	---	---	--	--	---	---	---

You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm to the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

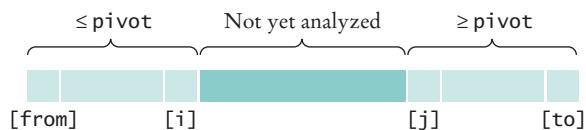
1	2	3	3	4			5	6	7
---	---	---	---	---	--	--	---	---	---

Quicksort is implemented recursively as follows:

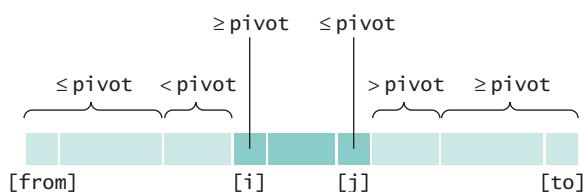
```
public static void sort(int[] a, int from, int to)
{
    if (from >= to) { return; }
    int p = partition(a, from, to);
    sort(a, from, p);
    sort(a, p + 1, to);
}
```

Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range, $a[from]$, as the pivot.

Now form two regions $a[from] \dots a[i]$, consisting of values at most as large as the pivot and $a[j] \dots a[to]$, consisting of values at least as large as the pivot. The region $a[i + 1] \dots a[j - 1]$ consists of values that haven't been analyzed yet. (See the figure below.) At the beginning, both the left and right areas are empty; that is, $i = from - 1$ and $j = to + 1$.

Partitioning a Range

Then keep incrementing i while $a[i] < \text{pivot}$ and keep decrementing j while $a[j] > \text{pivot}$. The figure below shows i and j when that process stops.

Extending the Partitions

Now swap the values in positions i and j , increasing both areas once more. Keep going while $i < j$. Here is the code for the partition method:

```
private static int partition(int[] a, int from, int to)
{
    int pivot = a[from];
    int i = from - 1;
    int j = to + 1;
    while (i < j)
    {
        i++; while (a[i] < pivot) { i++; }
        j--; while (a[j] > pivot) { j--; }
        if (i < j) { ArrayUtil.swap(a, i, j); }
    }
    return j;
}
```

On average, the quicksort algorithm is an $O(n \log(n))$ algorithm. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* run-time behavior is $O(n^2)$. Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such “tuned” quicksort algorithms are commonly used because their performance is generally excellent. For example, the sort method in the Arrays class uses a quicksort algorithm.

Another improvement that is commonly made in practice is to switch to insertion sort when the array is short, because the total number of operations using insertion sort is lower for short arrays. The Java library makes that switch if the array length is less than seven.

FULL CODE EXAMPLE

Go to wiley.com/go/bjlo2code to download a program that demonstrates the quicksort algorithm.

In quicksort, one partitions the elements into two groups, holding the smaller and larger elements. Then one sorts each group.



© Christopher Futchik/istockphoto.

14.6 Searching

Searching for an element in an array is an extremely common task. As with sorting, the right choice of algorithms can make a big difference.

14.6.1 Linear Search

Suppose you need to find your friend's telephone number. You look up the friend's name in the telephone book, and naturally you can find it quickly, because the telephone book is sorted alphabetically. Now suppose you have a telephone number and you must know to what party it belongs. You could of course call that number, but suppose nobody picks up on the other end. You could look through the telephone book, a number at a time, until you find the number. That would obviously be a tremendous amount of work, and you would have to be desperate to attempt it.

This thought experiment shows the difference between a search through an unsorted data set and a search through a sorted data set. The following two sections will analyze the difference formally.

If you want to find a number in a sequence of values that occur in arbitrary order, there is nothing you can do to speed up the search. You must simply look through all elements until you have found a match or until you reach the end. This is called a **linear** or **sequential** search.

How long does a linear search take? If we assume that the element v is present in the array a , then the average search visits $n/2$ elements, where n is the length of the array. If it is not present, then all n elements must be inspected to verify the absence. Either way, a linear search is an $O(n)$ algorithm.

Here is a class that performs linear searches through an array a of integers. When searching for a value, the search method returns the first index of the match, or -1 if the value does not occur in a .

A linear search examines all values in an array until it finds a match or reaches the end.

A linear search locates a value in an array in $O(n)$ steps.

sec06_01/LinearSearcher.java

```

1  /**
2   * A class for executing linear searches in an array.
3  */
4  public class LinearSearcher
5  {
6      /**
7       * Finds a value in an array, using the linear search
8       * algorithm.
9       * @param a the array to search
10      * @param value the value to find
11      * @return the index at which the value occurs, or -1
12      * if it does not occur in the array
13     */
14    public static int search(int[] a, int value)
15    {
16        for (int i = 0; i < a.length; i++)
17        {
18            if (a[i] == value) { return i; }
19        }
20        return -1;
}

```

```
21     }
22 }
```

sec06_01/LinearSearchDemo.java

```

1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 /**
5  * This program demonstrates the linear search algorithm.
6 */
7 public class LinearSearchDemo
8 {
9     public static void main(String[] args)
10    {
11        int[] a = ArrayUtil.randomIntArray(20, 100);
12        System.out.println(Arrays.toString(a));
13        Scanner in = new Scanner(System.in);
14
15        boolean done = false;
16        while (!done)
17        {
18            System.out.print("Enter number to search for, -1 to quit: ");
19            int n = in.nextInt();
20            if (n == -1)
21            {
22                done = true;
23            }
24            else
25            {
26                int pos = LinearSearcher.search(a, n);
27                System.out.println("Found in position " + pos);
28            }
29        }
30    }
31 }
```

Program Run

```
[46, 99, 45, 57, 64, 95, 81, 69, 11, 97, 6, 85, 61, 88, 29, 65, 83, 88, 45, 88]
Enter number to search for, -1 to quit: 12
Found in position -1
Enter number to search for, -1 to quit: -1
```

14.6.2 Binary Search

Now let us search for an item in a data sequence that has been previously sorted. Of course, we could still do a linear search, but it turns out we can do much better than that.

Consider the following sorted array a. The data set is:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	4	5	8	9	12	17	20	24	32

We would like to see whether the value 15 is in the data set. Let's narrow our search by finding whether the value is in the first or second half of the array. The last value

in the first half of the data set, $a[4]$, is 9, which is smaller than the value we are looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	4	5	8	9	12	17	20	24	32

The middle element of this sequence is 20; hence, the value must be located in the sequence:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	4	5	8	9	12	17	20	24	32

The last value of the first half of this very short sequence is 12, which is smaller than the value that we are searching, so we must look in the second half:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
1	4	5	8	9	12	17	20	24	32

A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.

It is trivial to see that we don't have a match, because $15 \neq 17$. If we wanted to insert 15 into the sequence, we would need to insert it just before $a[6]$.

This search process is called a **binary search**, because we cut the size of the search in half in each step. That cutting in half works only because we know that the sequence of values is sorted.

The following class implements binary searches in a sorted array of integers. The search method returns the position of the match if the search succeeds, or -1 if the value is not found in a . Here, we show a recursive version of the binary search algorithm.

sec06_02/BinarySearcher.java

```

1  /**
2   * A class for executing binary searches in an array.
3   */
4  public class BinarySearcher
5  {
6      /**
7       * Finds a value in a range of a sorted array, using the binary
8       * search algorithm.
9       * @param a the array in which to search
10      * @param low the low index of the range
11      * @param high the high index of the range
12      * @param value the value to find
13      * @return the index at which the value occurs, or -1
14      * if it does not occur in the array
15     */
16    public int search(int[] a, int low, int high, int value)
17    {
18        if (low <= high)
19        {
20            int mid = (low + high) / 2;
21
22            if (a[mid] == value)
23            {
24                return mid;
25            }
26            else if (a[mid] < value )
27            {

```

```

28         return search(a, mid + 1, high, value);
29     }
30     else
31     {
32         return search(a, low, mid - 1, value);
33     }
34 }
35 else
36 {
37     return -1;
38 }
39 }
40 }
```

Now let's determine the number of visits to array elements required to carry out a binary search. We can use the same technique as in the analysis of merge sort. Because we look at the middle element, which counts as one visit, and then search either the left or the right subarray, we have

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Using the same equation,

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1$$

By plugging this result into the original equation, we get

$$T(n) = T\left(\frac{n}{4}\right) + 2$$

That generalizes to

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

As in the analysis of merge sort, we make the simplifying assumption that n is a power of 2, $n = 2^m$, where $m = \log_2(n)$. Then we obtain

$$T(n) = 1 + \log_2(n)$$

Therefore, binary search is an $O(\log(n))$ algorithm.

That result makes intuitive sense. Suppose that n is 100. Then after each search, the size of the search range is cut in half, to 50, 25, 12, 6, 3, and 1. After seven comparisons we are done. This agrees with our formula, because $\log_2(100) \approx 6.64386$, and indeed the next larger power of 2 is $2^7 = 128$.

Because a binary search is so much faster than a linear search, is it worthwhile to sort an array first and then use a binary search? It depends. If you search the array only once, then it is more efficient to pay for an $O(n)$ linear search than for an $O(n \log(n))$ sort and an $O(\log(n))$ binary search. But if you will be making many searches in the same array, then sorting it is definitely worthwhile.

A binary search locates a value in a sorted array in $O(\log(n))$ steps.



- 18.** Suppose you need to look through 1,000,000 records to find a telephone number. How many records do you expect to search before finding the number?
- 19.** Why can't you use a "for each" loop for (int element : a) in the search method?
- 20.** Suppose you need to look through a sorted array with 1,000,000 elements to find a value. Using the binary search algorithm, how many records do you expect to search before finding the value?

Practice It Now you can try these exercises at the end of the chapter: R14.14, E14.13, E14.15.



Before pocket calculators and personal computers existed, navigators and engineers used mechanical adding machines, slide rules, and tables of logarithms and trigonometric functions to speed up computations. Unfortunately, the tables—for which values had to be computed by hand—were notoriously inaccurate. The mathematician Charles Babbage (1791–1871) had the insight that if a machine could be constructed that produced printed tables automatically, both calculation and typesetting errors could be avoided. Babbage set out to develop a machine for this purpose, which he called a *Difference Engine*.



Topham/The Image Works

Replica of Babbage's Difference Engine

Computing & Society 14.1 The First Programmer

because it used successive differences to compute polynomials. For example, consider the function $f(x) = x^3$. Write down the values for $f(1), f(2), f(3)$, and so on. Then take the *differences* between successive values:

1
7
8
19
27
37
64
61
125
91
216

Repeat the process, taking the difference of successive values in the second column, and then repeat once again:

1
7
8
12
19
6
27
18
6
37
6
64
24
6
61
6
125
30
91
216

Now the differences are all the same. You can retrieve the function values by a pattern of additions—you need to know the values at the fringe of the pattern and the constant difference. You can try it out yourself: Write the highlighted numbers on a sheet of paper and fill in the others by adding the numbers that are in the north and northwest positions.

This method was very attractive, because mechanical addition machines had been known for some time. They consisted of cog wheels, with 10 cogs per wheel, to represent digits, and mechanisms to handle the carry from one digit to the next. Mechanical multiplication machines, on the other hand, were fragile and unreliable. Babbage built a successful prototype of the Difference Engine and, with his own money and government grants, proceeded to build the table-printing machine. However, because of funding problems and the difficulty of building the machine to the required precision, it was never completed.

While working on the Difference Engine, Babbage conceived of a much grander vision that he called the *Analytical Engine*. The Difference Engine was designed to carry out a limited set of computations—it was no smarter than a pocket calculator is today. But Babbage realized that such a machine could be made *programmable* by storing programs as well as data. The internal storage of the Analytical Engine was to consist of 1,000 registers of 50 decimal digits each. Programs and constants were to be stored on punched cards—a technique that was, at that time, commonly used on looms for weaving patterned fabrics.

Ada Augusta, Countess of Lovelace (1815–1852), the only child of Lord Byron, was a friend and sponsor of Charles Babbage. Ada Lovelace was one of the first people to realize the potential of such a machine, not just for computing mathematical tables but for processing data that were not numbers. She is considered by many to be the world's first programmer.

14.7 Problem Solving: Estimating the Running Time of an Algorithm

In this chapter, you have learned how to estimate the running time of sorting algorithms. As you have seen, being able to differentiate between $O(n \log(n))$ and $O(n^2)$ running times has great practical implications. Being able to estimate the running times of other algorithms is an important skill. In this section, we will practice estimating the running time of array algorithms.

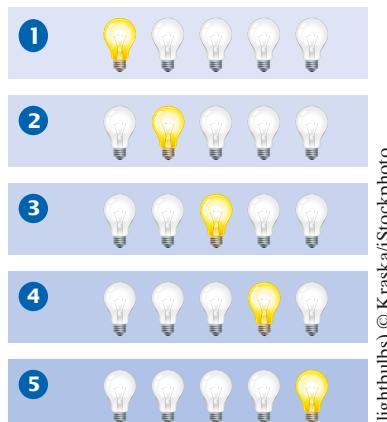
14.7.1 Linear Time

Let us start with a simple example, an algorithm that counts how many elements have a particular value:

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    if (a[i] == value) { count++; }
```

What is the running time in terms of n , the length of the array?

Start with looking at the pattern of array element visits. Here, we visit each element once. It helps to visualize this pattern. Imagine the array as a sequence of light bulbs. As the i th element gets visited, imagine the i th bulb lighting up.



(lightbulbs) © KraskaiStockphoto.

Now look at the work per visit. Does each visit involve a fixed number of actions, independent of n ? In this case, it does. There are just a few actions—read the element, compare it, maybe increment a counter.

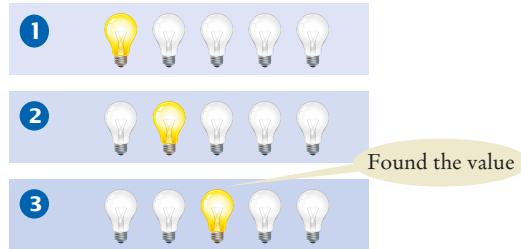
Therefore, the running time is n times a constant, or $O(n)$.

What if we don't always run to the end of the array? For example, suppose we want to check whether the value occurs in the array, without counting it:

```
boolean found = false;
for (int i = 0; !found && i < a.length; i++)
{
    if (a[i] == value) { found = true; }
```

A loop with n iterations has $O(n)$ running time if each step consists of a fixed number of actions.

Then the loop can stop in the middle:



Is this still $O(n)$? It is, because in some cases the match may be at the very end of the array. Also, if there is no match, one must traverse the entire array.

14.7.2 Quadratic Time

Now let's turn to a more interesting case. What if we do a lot of work with each visit? Here is an example: We want to find the most frequent element in an array.

Suppose the array is

8	7	5	7	7	5	4
---	---	---	---	---	---	---

It's obvious by looking at the values that 7 is the most frequent one. But now imagine an array with a few thousand values.

8	7	5	7	7	5	4	1	2	3	3	4	9	12	3	2	5	...	11	9	2	3	7	8
---	---	---	---	---	---	---	---	---	---	---	---	---	----	---	---	---	-----	----	---	---	---	---	---

We can count how often the value 8 occurs, then move on to count how often 7 occurs, and so on. For example, in the first array, 8 occurs once, and 7 occurs three times. Where do we put the counts? Let's put them into a second array of the same length.

a:	8	7	5	7	7	5	4
counts:	1	3	2	3	3	2	1

Then we take the maximum of the counts. It is 3. We look up where the 3 occurs in the counts, and find the corresponding value. Thus, the most common value is 7.

Let us first estimate how long it takes to compute the counts.

```
for (int i = 0; i < a.length; i++)
{
    counts[i] = Count how often a[i] occurs in a
}
```

A loop with n iterations has $O(n^2)$ running time if each step takes $O(n)$ time.

We still visit each array element once, but now the work per visit is much larger. As you have seen in the previous section, each counting action is $O(n)$. When we do $O(n)$ work in each step, the total running time is $O(n^2)$.

This algorithm has three phases:

1. Compute all counts.
2. Compute the maximum.
3. Find the maximum in the counts.

We have just seen that the first phase is $O(n^2)$. Computing the maximum is $O(n)$ —look at the algorithm in Section 6.3.3 and note that each step involves a fixed amount of work. Finally, we just saw that finding a value is $O(n)$.

The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.

How can we estimate the total running time from the estimates of each phase? Of course, the total time is the sum of the individual times, but for big-Oh estimates, we take the *maximum* of the estimates. To see why, imagine that we had actual equations for each of the times:

$$T_1(n) = an^2 + bn + c$$

$$T_2(n) = dn + e$$

$$T_3(n) = fn + g$$

Then the sum is

$$T(n) = T_1(n) + T_2(n) + T_3(n) = an^2 + (b + d + f)n + c + e + g$$

But only the largest term matters, so $T(n)$ is $O(n^2)$.

Thus, we have found that our algorithm for finding the most frequent element is $O(n^2)$.

14.7.3 The Triangle Pattern

Let us see if we can speed up the algorithm from the preceding section. It seems wasteful to count elements again if we have already counted them.

Can we save time by eliminating repeated counting of the same element? That is, before counting $a[i]$, should we first check that it didn't occur in $a[0] \dots a[i - 1]$?

Let us estimate the cost of these additional checks. In the i th step, the amount of work is proportional to i . That's not quite the same as in the preceding section, where you saw that a loop with n iterations, each of which takes $O(n)$ time, is $O(n^2)$. Now each step just takes $O(i)$ time.

To get an intuitive feel for this situation, look at the light bulbs again. In the second iteration, we visit $a[0]$ again. In the third iteration, we visit $a[0]$ and $a[1]$ again, and so on. The light bulb pattern is



A loop with n iterations has $O(n^2)$ running time if the i th step takes $O(i)$ time.

If there are n light bulbs, about half of the square above, or $n^2/2$ of them, light up. That's unfortunately still $O(n^2)$.

Here is another idea for time saving. When we count $a[i]$, there is no need to do the counting in $a[0] \dots a[i - 1]$. If $a[i]$ never occurred before, we get an accurate

count by just looking at $a[i] \dots a[n - 1]$. And if it did, we already have an accurate count. Does that help us? Not really—it's the triangle pattern again, but this time in the other direction.



That doesn't mean that these improvements aren't worthwhile. If an $O(n^2)$ algorithm is the best one can do for a particular problem, you still want to make it as fast as possible. However, we will not pursue this plan further because it turns out that we can do much better.

14.7.4 Logarithmic Time

An algorithm that cuts the size of work in half in each step runs in $O(\log(n))$ time.

Logarithmic time estimates arise from algorithms that cut work in half in each step. You have seen this in the algorithms for binary search and merge sort.

In particular, when you use sorting or binary search in a phase of an algorithm, you will encounter logarithmic time in the big-Oh estimates.

Consider this idea for improving our algorithm for finding the most frequent element. Suppose we first *sort* the array:

8 7 5 7 7 5 4 → 4 5 5 7 7 7 8

That cost us $O(n \log(n))$ time. If we can complete the algorithm in $O(n)$ time, we will have found a better algorithm than the $O(n^2)$ algorithm of the preceding sections.

To see why this is possible, imagine traversing the sorted array. As long as you find a value that was equal to its predecessor, you increment a counter. When you find a different value, save the counter and start counting anew:

a:	4 5 5 7 7 7 8
counts:	1 1 2 1 2 3 1

Or in code,

```
int count = 0;
for (int i = 0; i < a.length; i++)
{
    count++;
    if (i == a.length - 1 || a[i] != a[i + 1])
    {
        // Process count
    }
}
```

```

        counts[i] = count;
        count = 0;
    }
}

```

That's a constant amount of work per iteration, even though it visits two elements:



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a program for comparing the speed of algorithms that find the most frequent element.



SELF CHECK

$2n$ is still $O(n)$. Thus, we can compute the counts in $O(n)$ time from a sorted array. The entire algorithm is now $O(n \log(n))$.

Note that we don't actually need to keep all counts, only the highest one that we encountered so far (see Exercise E14.11). That is a worthwhile improvement, but it does not change the big-Oh estimate of the running time.

21. What is the “light bulb pattern” of visits in the following algorithm to check whether an array is a palindrome?


```

for (int i = 0; i < a.length / 2; i++)
{
    if (a[i] != a[a.length - 1 - i]) { return false; }
}
return true;
      
```
22. What is the big-Oh running time of the following algorithm to check whether the first element is duplicated in an array?


```

for (int i = 1; i < a.length; i++)
{
    if (a[0] == a[i]) { return true; }
}
return false;
      
```
23. What is the big-Oh running time of the following algorithm to check whether an array has a duplicate value?


```

for (int i = 0; i < a.length; i++)
{
    for (j = i + 1; j < a.length; j++)
    {
        if (a[i] == a[j]) { return true; }
    }
}
return false;
      
```
24. Describe an $O(n \log(n))$ algorithm for checking whether an array has duplicates.

25. What is the big-Oh running time of the following algorithm to find an element in an $n \times n$ array?

```
for (int i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        if (a[i][j] == value) { return true; }
    }
}
return false;
```

26. If you apply the algorithm of Section 14.7.4 to an $n \times n$ array, what is the big-Oh efficiency of finding the most frequent element in terms of n ?

Practice It Now you can try these exercises at the end of the chapter: R14.9, R14.15, R14.21, E14.11.

14.8 Sorting and Searching in the Java Library

When you write Java programs, you don't have to implement your own sorting algorithms. The `Arrays` and `Collections` classes provide sorting and searching methods that we will introduce in the following sections.

14.8.1 Sorting

The `Arrays` class implements a sorting method that you should use for your Java programs.

The `Arrays` class contains static sort methods to sort arrays of integers and floating-point numbers. For example, you can sort an array of integers simply as

```
int[] a = . . .;
Arrays.sort(a);
```

That `sort` method uses the quicksort algorithm—see Special Topic 14.3 for more information about that algorithm.

If your data are contained in an `ArrayList`, use the `Collections.sort` method instead; it uses the merge sort algorithm:

```
ArrayList<String> names = . . .;
Collections.sort(names);
```

The `Collections` class contains a sort method that can sort array lists.

14.8.2 Binary Search

The `Arrays` and `Collections` classes contain static `binarySearch` methods that implement the binary search algorithm, but with a useful enhancement. If a value is not found in the array, then the returned value is not -1 , but $-k - 1$, where k is the position before which the element should be inserted. For example,

```
int[] a = { 1, 4, 9 };
int v = 7;
int pos = Arrays.binarySearch(a, v);
// Returns -3; v should be inserted before position 2
```

14.8.3 Comparing Objects

The sort method of the Arrays class sorts objects of classes that implement the Comparable interface.

In application programs, you often need to sort or search through collections of objects. Therefore, the `Arrays` and `Collections` classes also supply `sort` and `binarySearch` methods for objects. However, these methods cannot know how to compare arbitrary objects. Suppose, for example, that you have an array of `Country` objects. It is not obvious how the countries should be sorted. Should they be sorted by their names or by their areas? The `sort` and `binarySearch` methods cannot make that decision for you. Instead, they require that the objects belong to classes that implement the `Comparable` interface type that was introduced in Section 9.6.3. That interface has a single method:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The call

```
a.compareTo(b)
```

must return a negative number if `a` should come before `b`, 0 if `a` and `b` are the same, and a positive number otherwise.

Note that `Comparable` is a generic type, similar to the `ArrayList` type. With an `ArrayList`, the type parameter denotes the type of the elements. With `Comparable`, the type parameter is the type of the parameter of the `compareTo` method. Therefore, a class that implements `Comparable` will want to be “comparable to itself”. For example, the `Country` class implements `Comparable<Country>`.

Many classes in the standard Java library, including the `String` class, number wrappers, dates, and file paths, implement the `Comparable` interface.

You can implement the `Comparable` interface for your own classes as well. For example, to sort a collection of countries by area, the `Country` class would implement the `Comparable<Country>` interface and provide a `compareTo` method like this:

```
public class Country implements Comparable<Country>
{
    public int compareTo(Country other)
    {
        return Double.compare(area, other.area);
    }
}
```

The `compareTo` method compares countries by their area. Note the use of the helper method `Double.compare` (see Programming Tip 9.2) that returns a negative integer, 0, or a positive integer. This is easier than programming a three-way branch.

Now you can pass an array of countries to the `Arrays.sort` method:

```
Country[] countries = new Country[n];
// Add countries
Arrays.sort(countries); // Sorts by increasing area
```

Whenever you need to carry out sorting or searching, use the methods in the `Arrays` and `Collections` classes and not those that you write yourself. The library algorithms have been fully debugged and optimized. Thus, the primary purpose of this chapter was not to teach you how to implement practical sorting and searching algorithms. Instead, you have learned something more important, namely that different algorithms can vary widely in performance, and that it is worthwhile to learn more about the design and analysis of algorithms.



FULL CODE EXAMPLE

Go to wiley.com/go/bj102code to download a program that demonstrates the Java library methods for sorting and searching.



- 27.** Why can't the `Arrays.sort` method sort an array of `Rectangle` objects?
- 28.** What steps would you need to take to sort an array of `BankAccount` objects by increasing balance?
- 29.** Why is it useful that the `Arrays.binarySearch` method indicates the position where a missing element should be inserted?
- 30.** Why does `Arrays.binarySearch` return $-k - 1$ and not $-k$ to indicate that a value is not present and should be inserted before position k ?

Practice It Now you can try these exercises at the end of the chapter: E14.12, E14.16, E14.17.

Common Error 14.1



The `compareTo` Method Can Return Any Integer, Not Just `-1`, `0`, and `1`

The call `a.compareTo(b)` is allowed to return *any* negative integer to denote that `a` should come before `b`, not necessarily the value `-1`. That is, the test

```
if (a.compareTo(b) == -1) // Error!
```

is generally wrong. Instead, you should test

```
if (a.compareTo(b) < 0) // OK
```

Why would a `compareTo` method ever want to return a number other than `-1`, `0`, or `1`? Sometimes, it is convenient to just return the difference of two integers. For example, the `compareTo` method of the `String` class compares characters in matching positions:

```
char c1 = charAt(i);
char c2 = other.charAt(i);
```

If the characters are different, then the method simply returns their difference:

```
if (c1 != c2) { return c1 - c2; }
```

This difference is a negative number if `c1` is less than `c2`, but it is not necessarily the number `-1`.

Note that returning a difference only works if it doesn't overflow (see Programming Tip 9.2).

Special Topic 14.4



The Comparator Interface

Sometimes you want to sort an array or array list of objects, but the objects don't belong to a class that implements the `Comparable` interface. Or, perhaps, you want to sort the array in a different order. For example, you may want to sort countries by name rather than by value.

You wouldn't want to change the implementation of a class simply to call `Arrays.sort`. Fortunately, there is an alternative. One version of the `Arrays.sort` method does not require that the objects belong to classes that implement the `Comparable` interface. Instead, you can supply arbitrary objects. However, you must also provide a *comparator* object whose job is to compare objects. The comparator object must belong to a class that implements the `Comparator` interface. That interface has a single method, `compare`, which compares two objects.

Just like `Comparable`, the `Comparator` interface is a parameterized type. The type parameter specifies the type of the `compare` parameter variables. For example, `Comparator<Country>` looks like this:

```
public interface Comparator<Country>
{
    int compare(Country a, Country b);
}
```

The call

```
comp.compare(a, b)
```

must return a negative number if a should come before b, 0 if a and b are the same, and a positive number otherwise. (Here, comp is an object of a class that implements Comparator<Country>.)

For example, here is a Comparator class for country:

```
public class CountryComparator implements Comparator<Country>
{
    public int compare(Country a, Country b)
    {
        return Double.compare(a.getArea(), b.getArea());
    }
}
```

To sort an array of countries by area, call

```
Arrays.sort(countries, new CountryComparator());
```

Java 8 Note 14.1



Comparators with Lambda Expressions

Before Java 8, it was cumbersome to specify a comparator. You had to come up with a class that implements the Comparator interface, implement the compare method, and construct an object of that class. That was unfortunate because comparators are very useful for several algorithms, such as searching, sorting, and finding the maximum or minimum.

With lambda expressions, it is easier to specify a comparator. For example, to sort an array of words by increasing lengths, call

```
Arrays.sort(words, (v, w) -> v.length() - w.length());
```

There is a convenient shortcut for this case. Note that the comparison depends on a function that maps each string to a numeric value, namely its length. The static method Comparator.comparing constructs a comparator from a lambda expression. For example, you can call

```
Arrays.sort(words, Comparator.comparing(w -> w.length()));
```

A comparator is constructed that calls the supplied function on both objects that are to be compared, and then compares the function results.

The Comparator.comparing method takes care of many common cases. For example, to sort countries by area, call

```
Arrays.sort(countries, Comparator.comparing(c -> c.getArea()));
```



WORKED EXAMPLE 14.1



Enhancing the Insertion Sort Algorithm

Learn how to implement an improvement of the insertion sort algorithm shown in Special Topic 14.2. The enhanced algorithm is called *Shell sort* after its inventor, Donald Shell. Go to wiley.com/go/bjlo2examples and download Worked Example 14.1.

CHAPTER SUMMARY

Describe the selection sort algorithm.



- The selection sort algorithm sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front.

Measure the running time of a method.

- To measure the running time of a method, get the current time immediately before and after the method call.

Use the big-Oh notation to describe the running time of an algorithm.

- Computer scientists use the big-Oh notation to describe the growth rate of a function.
- Selection sort is an $O(n^2)$ algorithm. Doubling the data set means a fourfold increase in processing time.
- Insertion sort is an $O(n^2)$ algorithm.



Describe the merge sort algorithm.



- The merge sort algorithm sorts an array by cutting the array in half, recursively sorting each half, and then merging the sorted halves.

Contrast the running times of the merge sort and selection sort algorithms.

- Merge sort is an $O(n \log(n))$ algorithm. The $n \log(n)$ function grows much more slowly than n^2 .

Describe the running times of the linear search algorithm and the binary search algorithm.

- A linear search examines all values in an array until it finds a match or reaches the end.
- A linear search locates a value in an array in $O(n)$ steps.
- A binary search locates a value in a sorted array by determining whether the value occurs in the first or second half, then repeating the search in one of the halves.
- A binary search locates a value in a sorted array in $O(\log(n))$ steps.

Practice developing big-Oh estimates of algorithms.

- A loop with n iterations has $O(n)$ running time if each step consists of a fixed number of actions.
- A loop with n iterations has $O(n^2)$ running time if each step takes $O(n)$ time.
- The big-Oh running time for doing several steps in a row is the largest of the big-Oh times for each step.



- A loop with n iterations has $O(n^2)$ running time if the i th step takes $O(i)$ time.
- An algorithm that cuts the size of work in half in each step runs in $O(\log(n))$ time.

Use the Java library methods for sorting and searching data.

- The `Arrays` class implements a sorting method that you should use for your Java programs.
- The `Collections` class contains a `sort` method that can sort array lists.
- The `sort` method of the `Arrays` class sorts objects of classes that implement the `Comparable` interface.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

`java.lang.System`
`currentTimeMillis`
`java.util.Arrays`
`binarySearch`
`sort`

`java.util.Collections`
`binarySearch`
`sort`
`java.util.Comparator<T>`
`compare`
`comparing`

REVIEW EXERCISES

- **R14.1** What is the difference between searching and sorting?
- **R14.2** *Checking against off-by-one errors.* When writing the selection sort algorithm of Section 14.1, a programmer must make the usual choices of `<` versus `<=`, `a.length` versus `a.length - 1`, and `from` versus `from + 1`. This is fertile ground for off-by-one errors. Conduct code walkthroughs of the algorithm with arrays of length 0, 1, 2, and 3 and check carefully that all index values are correct.
- **R14.3** For the following expressions, what is the order of the growth of each?

a. $n^2 + 2n + 1$ b. $n^{10} + 9n^9 + 20n^8 + 145n^7$ c. $(n+1)^4$ d. $(n^2 + n)^2$ e. $n + 0.001n^3$ f. $n^3 - 1000n^2 + 10^9$	g. $n + \log(n)$ h. $n^2 + n \log(n)$ i. $2^n + n^2$ j. $\frac{n^3 + 2n}{n^2 + 0.75}$
--	--

- **R14.4** We determined that the actual number of visits in the selection sort algorithm is

$$T(n) = \frac{1}{2}n^2 + \frac{5}{2}n - 3$$

We characterized this method as having $O(n^2)$ growth. Compute the actual ratios

$$T(2,000)/T(1,000)$$

$$T(4,000)/T(1,000)$$

$$T(10,000)/T(1,000)$$

and compare them with

$$f(2,000)/f(1,000)$$

$$f(4,000)/f(1,000)$$

$$f(10,000)/f(1,000)$$

where $f(n) = n^2$.

- **R14.5** Suppose algorithm A takes five seconds to handle a data set of 1,000 records. If the algorithm A is an $O(n)$ algorithm, approximately how long will it take to handle a data set of 2,000 records? Of 10,000 records?
- ■ **R14.6** Suppose an algorithm takes five seconds to handle a data set of 1,000 records. Fill in the following table, which shows the approximate growth of the execution times depending on the complexity of the algorithm.

	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n \log(n))$	$O(2^n)$
1,000	5	5	5	5	5
2,000					
3,000		45			
10,000					

For example, because $3,000^2/1,000^2 = 9$, the algorithm would take nine times as long, or 45 seconds, to handle a data set of 3,000 records.

- ■ **R14.7** Sort the following growth rates from slowest to fastest growth.

$O(n)$	$O(\log(n))$	$O(2^n)$	$O(n\sqrt{n})$
$O(n^3)$	$O(n^2 \log(n))$	$O(\sqrt{n})$	$O(n^{\log(n)})$
$O(n^n)$	$O(n \log(n))$		

- **R14.8** What is the growth rate of the standard algorithm to find the minimum value of an array? Of finding both the minimum and the maximum?
- **R14.9** What is the big-Oh time estimate of the following method in terms of n , the length of a ? Use the “light bulb pattern” method of Section 14.7 to visualize your result.

```
public static void swap(int[] a)
{
    int i = 0;
    int j = a.length - 1;
    while (i < j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
        i++;
        j--;
    }
}
```

- **R14.10** A *run* is a sequence of adjacent repeated values (see Exercise R6.23). Describe an $O(n)$ algorithm to find the length of the longest run in an array.
- **R14.11** Consider the task of finding the most frequent element in an array of length n . Here are three approaches:
- Sort the array, then find the longest run.
 - Allocate an array of counters of the same size as the original array. For each element, traverse the array and count how many other elements are equal to it, updating its counter. Then find the maximum count.
 - Keep variables for the most frequent element that you have seen so far and its frequency. For each index i , check whether $a[i]$ occurs in $a[0] \dots a[i - 1]$. If not, count how often it occurs in $a[i + 1] \dots a[n - 1]$. If $a[i]$ is more frequent than the most frequent element so far, update the variables.
- Describe the big-Oh efficiency of each approach.
- **R14.12** Trace a walkthrough of selection sort with these sets:
- $4 \quad 7 \quad 11 \quad 4 \quad 9 \quad 5 \quad 11 \quad 7 \quad 3 \quad 5$
 - $-7 \quad 6 \quad 8 \quad 7 \quad 5 \quad 9 \quad 0 \quad 11 \quad 10 \quad 5 \quad 8$
- **R14.13** Trace a walkthrough of merge sort with these sets:
- $5 \quad 11 \quad 7 \quad 3 \quad 5 \quad 4 \quad 7 \quad 11 \quad 4 \quad 9$
 - $9 \quad 0 \quad 11 \quad 10 \quad 5 \quad 8 \quad -7 \quad 6 \quad 8 \quad 7 \quad 5$
- **R14.14** Trace a walkthrough of:
- Linear search for 7 in $-7 \quad 1 \quad 3 \quad 3 \quad 4 \quad 7 \quad 11 \quad 13$
 - Binary search for 8 in $-7 \quad 2 \quad 2 \quad 3 \quad 4 \quad 7 \quad 8 \quad 11 \quad 13$
 - Binary search for 8 in $-7 \quad 1 \quad 2 \quad 3 \quad 5 \quad 7 \quad 10 \quad 13$
- **R14.15** Your task is to remove all duplicates from an array. For example, if the array has the values
- $$4 \ 7 \ 11 \ 4 \ 9 \ 5 \ 11 \ 7 \ 3 \ 5$$
- then the array should be changed to
- $$4 \ 7 \ 11 \ 9 \ 5 \ 3$$
- Here is a simple algorithm: Look at $a[i]$. Count how many times it occurs in a . If the count is larger than 1, remove it. What is the growth rate of the time required for this algorithm?
- **R14.16** Modify the merge sort algorithm to remove duplicates in the merging step to obtain an algorithm that removes duplicates from an array. Note that the resulting array does not have the same ordering as the original one. What is the efficiency of this algorithm?
- **R14.17** Consider the following algorithm to remove all duplicates from an array: Sort the array. For each element in the array, look at its next neighbor to decide whether it is present more than once. If so, remove it. Is this a faster algorithm than the one in Exercise R14.15?
- **R14.18** Develop an $O(n \log(n))$ algorithm for removing duplicates from an array if the resulting array must have the same ordering as the original array. When a value occurs multiple times, all but its first occurrence should be removed.

- R14.19** Why does insertion sort perform significantly better than selection sort if an array is already sorted?
- R14.20** Consider the following speedup of the insertion sort algorithm of Special Topic 14.2. For each element, use the enhanced binary search algorithm that yields the insertion position for missing elements. Does this speedup have a significant impact on the efficiency of the algorithm?
- R14.21** Consider the following algorithm known as *bubble sort*:
- ```

While the array is not sorted
 For each adjacent pair of elements
 If the pair is not sorted
 Swap its elements.

```
- What is the big-Oh efficiency of this algorithm?
- R14.22** The *radix sort* algorithm sorts an array of  $n$  integers with  $d$  digits, using ten auxiliary arrays. First place each value  $v$  into the auxiliary array whose index corresponds to the last digit of  $v$ . Then move all values back into the original array, preserving their order. Repeat the process, now using the next-to-last (tens) digit, then the hundreds digit, and so on. What is the big-Oh time of this algorithm in terms of  $n$  and  $d$ ? When is this algorithm preferable to merge sort?
- R14.23** A *stable sort* does not change the order of elements with the same value. This is a desirable feature in many applications. Consider a sequence of e-mail messages. If you sort by date and then by sender, you'd like the second sort to preserve the relative order of the first, so that you can see all messages from the same sender in date order. Is selection sort stable? Insertion sort? Why or why not?
- R14.24** Give an  $O(n)$  algorithm to sort an array of  $n$  bytes (numbers between  $-128$  and  $127$ ). *Hint:* Use an array of counters.
- R14.25** You are given a sequence of arrays of words, representing the pages of a book. Your task is to build an index (a sorted array of words), each element of which has an array of sorted numbers representing the pages on which the word appears. Describe an algorithm for building the index and give its big-Oh running time in terms of the total number of words.
- R14.26** Given two arrays of  $n$  integers each, describe an  $O(n \log(n))$  algorithm for determining whether they have an element in common.
- R14.27** Given an array of  $n$  integers and a value  $v$ , describe an  $O(n \log(n))$  algorithm to find whether there are two values  $x$  and  $y$  in the array with sum  $v$ .
- R14.28** Given two arrays of  $n$  integers each, describe an  $O(n \log(n))$  algorithm for finding all elements that they have in common.
- R14.29** Suppose we modify the quicksort algorithm from Special Topic 14.3, selecting the middle element instead of the first one as pivot. What is the running time on an array that is already sorted?
- R14.30** Suppose we modify the quicksort algorithm from Special Topic 14.3, selecting the middle element instead of the first one as pivot. Find a sequence of values for which this algorithm has an  $O(n^2)$  running time.

## PRACTICE EXERCISES

- **E14.1** Modify the selection sort algorithm to sort an array of integers in descending order.
- **E14.2** Modify the selection sort algorithm to sort an array of coins by their value.
- **E14.3** Write a program that automatically generates the table of sample run times for the selection sort algorithm. The program should ask for the smallest and largest value of  $n$  and the number of measurements and then make all sample runs.
- **E14.4** Modify the merge sort algorithm to sort an array of strings in lexicographic order.
- **E14.5** Modify the selection sort algorithm to sort an array of objects that implement the `Measurable` interface from Chapter 9.
- **E14.6** Modify the selection sort algorithm to sort an array of objects that implement the `Comparable` interface (without a type parameter).
- **E14.7** Modify the selection sort algorithm to sort an array of objects, given a parameter of type `Comparator` (without a type parameter).
- **E14.8** Write a telephone lookup program. Read a data set of 1,000 names and telephone numbers from a file that contains the numbers in random order. Handle lookups by name and also reverse lookups by phone number. Use a binary search for both lookups.
- **E14.9** Implement a program that measures the performance of the insertion sort algorithm described in Special Topic 14.2.
- **E14.10** Implement the bubble sort algorithm described in Exercise R14.21.
- **E14.11** Implement the algorithm described in Section 14.7.4, but only remember the value with the highest frequency so far:
 

```
int mostFrequent = 0;
int highestFrequency = -1;
for (int i = 0; i < a.length; i++)
 Count how often a[i] occurs in a[i + 1] ... a[a.length - 1]
 If it occurs more often than highestFrequency
 highestFrequency = that count
 mostFrequent = a[i]
```
- **E14.12** Write a program that sorts an `ArrayList<Country>` in decreasing order so that the largest country is at the beginning of the array. Use a `Comparator`.
- **E14.13** Consider the binary search algorithm in Section 14.6. If no match is found, the search method returns  $-1$ . Modify the method so that if  $a$  is not found, the method returns  $-k - 1$ , where  $k$  is the position before which the element should be inserted. (This is the same behavior as `Arrays.binarySearch()`.)
- **E14.14** Implement the `sort` method of the merge sort algorithm without recursion, where the length of the array is a power of 2. First merge adjacent regions of size 1, then adjacent regions of size 2, then adjacent regions of size 4, and so on.
- **E14.15** Use insertion sort and the binary search from Exercise E14.13 to sort an array as described in Exercise R14.20. Implement this algorithm and measure its performance.

- **E14.16** Supply a class Person that implements the Comparable interface. Compare persons by their names. Ask the user to input ten names and generate ten Person objects. Using the compareTo method, determine the first and last person among them and print them.
- **E14.17** Sort an array list of strings by increasing length. Hint: Supply a Comparator.
- **E14.18** Sort an array list of strings by increasing length, and so that strings of the same length are sorted lexicographically. Hint: Supply a Comparator.

## PROGRAMMING PROJECTS

- **P14.1** It is common for people to name directories as dir1, dir2, and so on. When there are ten or more directories, the operating system displays them in dictionary order, as dir1, dir10, dir11, dir12, dir2, dir3, and so on. That is irritating, and it is easy to fix. Provide a comparator that compares strings that end in digit sequences in a way that makes sense to a human. First compare the part before the digits as strings, and then compare the numeric values of the digits.
- **P14.2** Sometimes, directory or file names have numbers in the middle, and there may be more than one number, for example, sec3\_14.txt or sec10\_1.txt. Provide a comparator that can compare such strings in a way that makes sense to humans. Break each string into strings not containing digits and digit groups. Then compare two strings by comparing the first non-digit groups as strings, the first digit groups as integers, and so on.
- **P14.3** The median  $m$  of a sequence of  $n$  elements is the element that would fall in the middle if the sequence was sorted. That is,  $e \leq m$  for half the elements, and  $m \leq e$  for the others. Clearly, one can obtain the median by sorting the sequence, but one can do quite a bit better with the following algorithm that finds the  $k$ th element of a sequence between  $a$  (inclusive) and  $b$  (exclusive). (For the median, use  $k = n / 2$ ,  $a = 0$ , and  $b = n$ .)

select(k, a, b):

Pick a pivot  $p$  in the subsequence between  $a$  and  $b$ .

Partition the subsequence elements into three subsequences: the elements  $<p, =p, >p$

Let  $n1, n2, n3$  be the sizes of each of these subsequences.

if  $k < n1$

    return select( $k, 0, n1$ ).

else if ( $k > n1 + n2$ )

    return select( $k, n1 + n2, n$ ).

else

    return  $p$ .

Implement this algorithm and measure how much faster it is for computing the median of a random large sequence, when compared to sorting the sequence and taking the middle element.

- **P14.4** Implement the following modification of the quicksort algorithm, due to Bentley and McIlroy. Instead of using the first element as the pivot, use an approximation of the median.

If  $n \leq 7$ , use the middle element. If  $n \leq 40$ , use the median of the first, middle, and last element. Otherwise compute the “pseudomedian” of the nine elements  $a[i * (n - 1) / 8]$ , where  $i$  ranges from 0 to 8. The pseudomedian of nine values is  $\text{med}(\text{med}(v_0, v_1, v_2), \text{med}(v_3, v_4, v_5), \text{med}(v_6, v_7, v_8))$ .

Compare the running time of this modification with that of the original algorithm on sequences that are nearly sorted or reverse sorted, and on sequences with many identical elements. What do you observe?

- **P14.5** Bentley and McIlroy suggest the following modification to the quicksort algorithm when dealing with data sets that contain many repeated elements.

Instead of partitioning as



(where  $\leq$  denotes the elements that are  $\leq$  the pivot), it is better to partition as



However, that is tedious to achieve directly. They recommend to partition as



and then swap the two  $=$  regions into the middle. Implement this modification and check whether it improves performance on data sets with many repeated elements.

- **P14.6** Implement the radix sort algorithm described in Exercise R14.22 to sort arrays of numbers between 0 and 999.
- **P14.7** Implement the radix sort algorithm described in Exercise R14.22 to sort arrays of numbers between 0 and 999. However, use a single auxiliary array, not ten.
- **P14.8** Implement the radix sort algorithm described in Exercise R14.22 to sort arbitrary int values (positive or negative).
- **P14.9** Implement the sort method of the merge sort algorithm without recursion, where the length of the array is an arbitrary number. Keep merging adjacent regions whose size is a power of 2, and pay special attention to the last area whose size is less.

## ANSWERS TO SELF-CHECK QUESTIONS

1. Dropping the `temp` variable would not work. Then  $a[i]$  and  $a[j]$  would end up being the same value.
2.  $1 | 5 4 3 2 6$   
   $1 2 | 4 3 5 6$   
   $1 2 3 4 5 6$
3. In each step, find the *maximum* of the remaining elements and swap it with the current element (or see Self Check 4).
4. The modified algorithm sorts the array in descending order.
5. Four times as long as 40,000 values, or about 37 seconds.
6. A parabola.
7. It takes about 100 times longer.
8. If  $n$  is 4, then  $\frac{1}{2}n^2$  is 8 and  $\frac{5}{2}n - 3$  is 7.
9. The first algorithm requires one visit, to store the new element. The second algorithm requires  $T(p) = 2 \times (n - p - 1)$  visits, where  $p$  is the location at which the element is removed. We don't know where that element is, but if elements are removed at random locations, on average, half of the removals will be above the middle and half below, so we can assume an average  $p$  of  $n / 2$  and  $T(n) = 2 \times (n - n / 2 - 1) = n - 2$ .

10. The first algorithm is  $O(1)$ , the second  $O(n)$ .
11. We need to check that  $a[0] \leq a[1]$ ,  $a[1] \leq a[2]$ , and so on, visiting  $2n - 2$  elements. Therefore, the running time is  $O(n)$ .
12. Let  $n$  be the length of the array. In the  $k$ th step, we need  $k$  visits to find the minimum. To remove it, we need an average of  $k - 2$  visits (see Self Check 9). One additional visit is required to add it to the end. Thus, the  $k$ th step requires  $2k - 1$  visits. Because  $k$  goes from  $n$  to 2, the total number of visits is

$$\begin{aligned}2n - 1 + 2(n - 1) - 1 + \dots + 2 \cdot 3 - 1 + 2 \cdot 2 - 1 = \\2(n + (n - 1) + \dots + 3 + 2 + 1 - 1) - (n - 1) = \\n(n + 1) - 2 - n + 1 = n^2 - 3\end{aligned}$$

(because  $1 + 2 + 3 + \dots + (n - 1) + n = n(n + 1)/2$ ). Therefore, the total number of visits is  $O(n^2)$ .

13. When the preceding `while` loop ends, the loop condition must be false, that is, `iFirst >= first.length` or `iSecond >= second.length` (De Morgan's Law).
14. First sort 8 7 6 5. Recursively, first sort 8 7. Recursively, first sort 8. It's sorted. Sort 7. It's sorted. Merge them: 7 8. Do the same with 6 5 to get 5 6. Merge them to 5 6 7 8. Do the same with 4 3 2 1: Sort 4 3 by sorting 4 and 3 and merging them to 3 4. Sort 2 1 by sorting 2 and 1 and merging them to 1 2. Merge 3 4 and 1 2 to 1 2 3 4. Finally, merge 5 6 7 8 and 1 2 3 4 to 1 2 3 4 5 6 7 8.
15. If the array size is 1, return its only element as the sum. Otherwise, recursively compute the sum of the first and second subarray and return the sum of these two values.
16. Approximately  $(100,000 \cdot \log(100,000)) / (50,000 \cdot \log(50,000)) = 2 \cdot 5 / 4.7 = 2.13$  times the time required for 50,000 values. That's  $2.13 \cdot 192$  milliseconds or approximately 409 milliseconds.
17.  $\frac{2n \log(2n)}{n \log(n)} = 2 \frac{1 + \log(n)}{\log(n)}$ .

For  $n > 2$ , that is a value  $< 3$ .

18. On average, you'd make 500,000 comparisons.
19. The search method returns the index at which the match occurs, not the data stored at that location.
20. You would search about 20. (The binary log of 1,024 is 10.)
21. 
22. It is an  $O(n)$  algorithm.
23. It is an  $O(n^2)$  algorithm—the number of visits follows a triangle pattern.
24. Sort the array, then make a linear scan to check for adjacent duplicates.
25. It is an  $O(n^2)$  algorithm—the outer and inner loops each have  $n$  iterations.
26. Because an  $n \times n$  array has  $m = n^2$  elements, and the algorithm in Section 14.7.4, when applied to an array with  $m$  elements, is  $O(m \log(m))$ , we have an  $O(n^2 \log(n))$  algorithm. Recall that  $\log(n^2) = 2 \log(n)$ , and the factor of 2 is irrelevant in the big-Oh notation.
27. The `Rectangle` class does not implement the `Comparable` interface.
28. The `BankAccount` class would need to implement the `Comparable` interface. Its `compareTo` method must compare the bank balances.
29. Then you know where to insert it so that the array stays sorted, and you can keep using binary search.
30. Otherwise, you would not know whether a value is present when the method returns 0.

## WORKED EXAMPLE 14.1

## Enhancing the Insertion Sort Algorithm



**Problem Statement** Implement an improvement of the insertion sort algorithm (in Special Topic 14.2) called *Shell sort* after its inventor, Donald Shell.

Shell sort is an enhancement of insertion sort that takes advantage of the fact that insertion sort is an  $O(n)$  algorithm if the array is already sorted. Shell sort brings parts of the array into sorted order, then runs an insertion sort over the entire array, so that the final sort doesn't do much work.

A key step in Shell sort is to arrange the sequence into rows and columns, and then to sort each column separately. For example, if the array is

|    |    |    |    |    |   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |
|----|----|----|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|
| 65 | 46 | 14 | 52 | 38 | 2 | 96 | 39 | 14 | 33 | 13 | 4 | 24 | 99 | 89 | 77 | 73 | 87 | 36 | 81 |
|----|----|----|----|----|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|

and we arrange it into four columns, we get

|    |    |    |    |
|----|----|----|----|
| 65 | 46 | 14 | 52 |
| 38 | 2  | 96 | 39 |
| 14 | 33 | 13 | 4  |
| 24 | 99 | 89 | 77 |
| 73 | 87 | 36 | 81 |

Now we sort each column:

|    |    |    |    |
|----|----|----|----|
| 14 | 2  | 13 | 5  |
| 24 | 33 | 14 | 39 |
| 38 | 46 | 36 | 52 |
| 65 | 87 | 89 | 77 |
| 73 | 99 | 96 | 81 |

Put together as a single array, we get

|    |   |    |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|----|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 14 | 2 | 13 | 5 | 24 | 33 | 14 | 39 | 38 | 46 | 36 | 52 | 65 | 87 | 89 | 77 | 73 | 99 | 96 | 81 |
|----|---|----|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

Note that the array isn't completely sorted, but many of the small numbers are now in front, and many of the large numbers are in the back.

We will repeat the process until the array is sorted. Each time, we use a different number of columns. Shell had originally used powers of two for the column counts. For example, on an array with 20 elements, he proposed using 16, 8, 4, 2, and finally one column. With one column, we have a plain insertion sort, so we know the array will be sorted. What is surprising is that the preceding sorts greatly speed up the process.

However, better sequences have been discovered. We will use the sequence of column counts

$$\begin{aligned} c_1 &= 1 \\ c_2 &= 4 \\ c_3 &= 13 \\ c_4 &= 40 \\ &\dots \\ c_{i+1} &= 3c_i + 1 \end{aligned}$$

That is, for an array with 20 elements, we first do a 13-sort, then a 4-sort, and then a 1-sort. This sequence is almost as good as the best known ones, and it is easy to compute.

We will not actually rearrange the array, but compute the locations of the elements of each column.

## WE2 Chapter 14 Sorting and Searching

For example, if the number of columns  $c$  is 4, the four columns are located in the array as follows:

|    |    |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|
| 65 |    | 38 |    | 14 |    | 24 |    | 73 |    |
|    | 46 |    | 2  |    | 33 |    | 99 |    | 87 |
|    |    | 14 |    | 96 |    | 13 |    | 89 |    |
|    |    |    | 52 |    | 39 |    | 4  |    | 36 |
|    |    |    |    |    |    |    |    | 77 |    |
|    |    |    |    |    |    |    |    |    | 81 |

Note that successive column elements have distance  $c$  from another. The  $k$ th column is made up of the elements  $a[k], a[k + c], a[k + 2 * c]$ , and so on.

Now let's adapt the insertion sort algorithm to sort such a column. The original algorithm was

```
for (int i = 1; i < a.length; i++)
{
 int next = a[i];
 // Move all larger elements up
 int j = i;
 while (j > 0 && a[j - 1] > next)
 {
 a[j] = a[j - 1];
 j--;
 }
 // Insert the element
 a[j] = next;
}
```

The outer loop visits the elements  $a[1], a[2]$ , and so on. In the  $k$ th column, the corresponding sequence is  $a[k + c], a[k + 2 * c]$ , and so on. That is, the outer loop becomes

```
for (int i = k + c; i < a.length; i = i + c)
```

In the inner loop, we originally visited  $a[j], a[j - 1]$ , and so on. We need to change that to  $a[j], a[j - c]$ , and so on. The inner loop becomes

```
while (j >= c && a[j - c] > next)
{
 a[j] = a[j - c];
 j = j - c;
}
```

Putting everything together, we get the following method:

```
/**
 * Sorts a column, using insertion sort.
 * @param a the array to sort
 * @param k the index of the first element in the column
 * @param c the gap between elements in the column
 */
public static void insertionSort(int[] a, int k, int c)
{
 for (int i = k + c; i < a.length; i = i + c)
 {
 int next = a[i];
 // Move all larger elements up
 int j = i;
 while (j >= c && a[j - c] > next)
 {
 a[j] = a[j - c];
 j = j - c;
 }
 }
}
```

```

 // Insert the element
 a[j] = next;
 }
}

```

Now we are ready to implement the Shell sort algorithm. First, we need to find out how many elements we need from the sequence of column counts. We generate the sequence values until they exceed the size of the array to be sorted.

```

ArrayList<Integer> columns = new ArrayList<Integer>();
int c = 1;
while (c < a.length)
{
 columns.add(c);
 c = 3 * c + 1;
}

```

For each column count, we sort all columns:

```

for (int s = columns.size() - 1; s >= 0; s--)
{
 c = columns.get(s);
 for (int k = 0; k < c; k++)
 {
 insertionSort(a, k, c);
 }
}

```

How good is the performance? Let's compare with the `Arrays.sort` method in the Java library.

```

int[] a = ArrayUtil.randomIntArray(n, 100);
int[] a2 = Arrays.copyOf(a, a.length);

StopWatch timer = new StopWatch();

timer.start();
ShellSorter.sort(a);
timer.stop();

System.out.println("Elapsed time with Shell sort: "
 + timer.getElapsedTime() + " milliseconds");

timer.reset();
timer.start();
Arrays.sort(a2);
timer.stop();

System.out.println("Elapsed time with Arrays.sort: "
 + timer.getElapsedTime() + " milliseconds");

if (!Arrays.equals(a, a2))
{
 throw new IllegalStateException("Incorrect sort result");
}

```

We make sure to sort the same array with both algorithms. Also, we check that the result of the Shell sort is correct by comparing it against the result of `Arrays.sort`.

Finally, we compare with the insertion sort algorithm.

The results show that Shell sort is a dramatic improvement over insertion sort:

```

Enter array size: 1000000
Elapsed time with Shell sort: 205 milliseconds

```

## WE4 Chapter 14 Sorting and Searching

```
Elapsed time with Arrays.sort: 101 milliseconds
Elapsed time with insertion sort: 148196 milliseconds
```

However, quicksort (which is used in `Arrays.sort`) outperforms Shell sort. For this reason, Shell sort is not used in practice, but it is still an interesting algorithm that is surprisingly effective.

You may also find it interesting to experiment with Shell's original column sizes. In the `sort` method, simply replace

```
c = 3 * c + 1;
```

with

```
c = 2 * c;
```

You will find that the algorithm is about three times slower than the improved sequence. That is still much faster than plain insertion sort.

You will find a program to demonstrate Shell sort and compare it to insertion sort in the `ch14/worked_example_1` folder of the book's companion code.

---

# THE JAVA COLLECTIONS FRAMEWORK

## CHAPTER GOALS

To learn how to use the collection classes supplied in the Java library

To use iterators to traverse collections

To choose appropriate collections for solving programming problems

To study applications of stacks and queues

## CHAPTER CONTENTS

### 15.1 AN OVERVIEW OF THE COLLECTIONS FRAMEWORK 692

### 15.2 LINKED LISTS 695

**C&S** Standardization 700

### 15.3 SETS 701

**PT1** Use Interface References to Manipulate Data Structures 705

### 15.4 MAPS 706

**J81** Updating Map Entries 708

**HT1** Choosing a Collection 708

**WE1** Word Frequency 

**ST1** Hash Functions 710



© nicholas belton/iStockphoto.

### 15.5 STACKS, QUEUES, AND PRIORITY QUEUES 712

### 15.6 STACK AND QUEUE APPLICATIONS 715

**WE2** Simulating a Queue of Waiting Customers 

**VE1** Building a Table of Contents 

**ST2** Reverse Polish Notation 723



© nicholas belton/iStockphoto.

If you want to write a program that collects objects (such as the stamps to the left), you have a number of choices. Of course, you can use an array list, but computer scientists have invented other mechanisms that may be better suited for the task. In this chapter, we introduce the collection classes and interfaces that the Java library offers. You will learn how to use the Java collection classes, and how to choose the most appropriate collection type for a problem.

## 15.1 An Overview of the Collections Framework

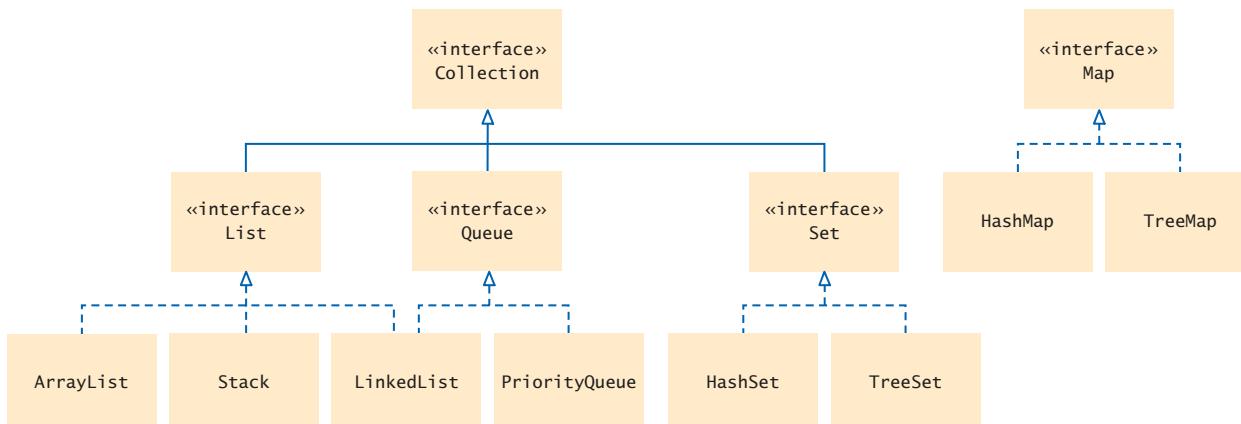
A collection groups together elements and allows them to be retrieved later.

When you need to organize multiple objects in your program, you can place them into a **collection**. The `ArrayList` class that was introduced in Chapter 6 is one of many collection classes that the standard Java library supplies. In this chapter, you will learn about the Java *collections framework*, a hierarchy of interface types and classes for collecting objects. Each interface type is implemented by one or more classes (see Figure 1).

At the root of the hierarchy is the `Collection` interface. That interface has methods for adding and removing elements, and so on. Table 1 on page 694 shows all the methods. Because all collections implement this interface, its methods are available for all collection classes. For example, the `size` method reports the number of elements in *any* collection.

The `List` interface describes an important category of collections. In Java, a *list* is a collection that remembers the order of its elements (see Figure 2). The `ArrayList` class implements the `List` interface. An `ArrayList` is simply a class containing an array that is expanded as needed. If you are not concerned about efficiency, you can use the `ArrayList` class whenever you need to collect objects. However, several common operations are inefficient with array lists. In particular, if an element is added or removed, the elements at larger positions must be moved.

The Java library supplies another class, `LinkedList`, that also implements the `List` interface. Unlike an array list, a linked list allows efficient insertion and removal of elements in the middle of the list. We will discuss that class in the next section.



**Figure 1** Interfaces and Classes in the Java Collections Framework

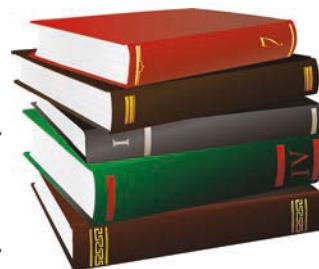


© Filip Fuxa/iStockphoto.

**Figure 2** A List of Books

© paremar/iStockphoto.

A list is a collection that remembers the order of its elements.

**Figure 3** A Set of Books

© Vladimir Tremin/iStockphoto.

**Figure 4** A Stack of Books

You use a list whenever you want to retain the order that you established. For example, on your bookshelf, you may order books by topic. A list is an appropriate data structure for such a collection because the ordering matters to you.

However, in many applications, you don't really care about the order of the elements in a collection. Consider a mail-order dealer of books. Without customers browsing the shelves, there is no need to order books by topic. Such a collection without an intrinsic order is called a **set**—see Figure 3.

Because a set does not track the order of the elements, it can arrange the elements so that the operations of finding, adding, and removing elements become more efficient. Computer scientists have invented mechanisms for this purpose. The Java library provides classes that are based on two such mechanisms (called *hash tables* and *binary search trees*). You will learn in this chapter how to choose between them.

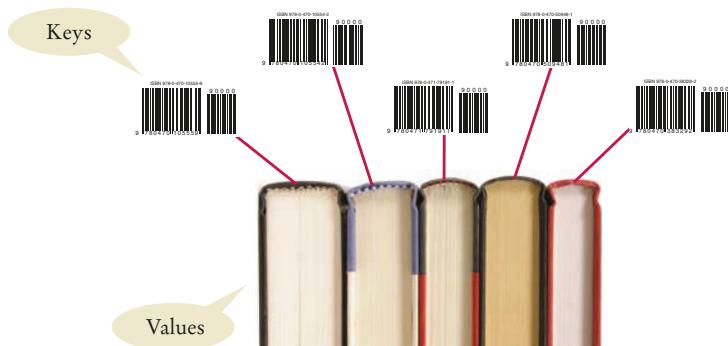
Another way of gaining efficiency in a collection is to reduce the number of operations. A **stack** remembers the order of its elements, but it does not allow you to insert elements in every position. You can add and remove elements only at the top—see Figure 4.

In a **queue**, you add items to one end (the tail) and remove them from the other end (the head). For example, you could keep a queue of books, adding required reading at the tail and taking a book from the head whenever you have time to read another one. A **priority queue** is an unordered collection that has an efficient operation for removing the element with the highest priority. You might use a priority queue for organizing your reading assignments. Whenever you have some time, remove the book with the highest priority and read it. We will discuss stacks, queues, and priority queues in Section 15.5.

Finally, a **map** manages associations between *keys* and *values*. Every key in the map has an associated value (see Figure 5). The map stores the keys, values, and the associations between them.

A set is an unordered collection of unique elements.

A map keeps associations between key and value objects.



**Figure 5**  
A Map from Bar Codes to Books

(books) © david franklin/iStockphoto.

**FULL CODE EXAMPLE**

Go to [wiley.com/go/bj102code](http://www.wiley.com/go/bj102code) to download a sample program that demonstrates several collection classes.

For an example, consider a library that puts a bar code on each book. The program used to check books in and out needs to look up the book associated with each bar code. A map associating bar codes with books can solve this problem. We will discuss maps in Section 15.4.

Starting with this chapter, we will use the “diamond syntax” for constructing instances of generic classes (see Special Topic 6.6). For example, when constructing an array list of strings, we will use

```
ArrayList<String> coll = new ArrayList<>();
```

Note that there is an empty pair of brackets `<>` after `new ArrayList` on the right-hand side. The compiler infers from the left-hand side that an array list of strings is constructed.

**Table 1** The Methods of the Collection Interface

|                                                                               |                                                                                                                              |
|-------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| <code>Collection&lt;String&gt; coll = new ArrayList&lt;&gt;();</code>         | The <code>ArrayList</code> class implements the <code>Collection</code> interface.                                           |
| <code>coll = new TreeSet&lt;&gt;();</code>                                    | The <code>TreeSet</code> class (Section 15.3) also implements the <code>Collection</code> interface.                         |
| <code>int n = coll.size();</code>                                             | Gets the size of the collection. <code>n</code> is now 0.                                                                    |
| <code>coll.add("Harry");<br/>coll.add("Sally");</code>                        | Adds elements to the collection.                                                                                             |
| <code>String s = coll.toString();</code>                                      | Returns a string with all elements in the collection. <code>s</code> is now [Harry, Sally].                                  |
| <code>System.out.println(coll);</code>                                        | Invokes the <code>toString</code> method and prints [Harry, Sally].                                                          |
| <code>coll.remove("Harry");<br/>boolean b = coll.remove("Tom");</code>        | Removes an element from the collection, returning <code>false</code> if the element is not present. <code>b</code> is false. |
| <code>b = coll.contains("Sally");</code>                                      | Checks whether this collection contains a given element. <code>b</code> is now true.                                         |
| <code>for (String s : coll)<br/>{<br/>    System.out.println(s);<br/>}</code> | You can use the “for each” loop with any collection. This loop prints the elements on separate lines.                        |
| <code>Iterator&lt;String&gt; iter = coll.iterator();</code>                   | You use an iterator for visiting the elements in the collection (see Section 15.2.3).                                        |

**SELF CHECK**

1. A grade book application stores a collection of quizzes. Should it use a list or a set?
2. A student information system stores a collection of student records for a university. Should it use a list or a set?
3. Why is a queue of books a better choice than a stack for organizing your required reading?
4. As you can see from Figure 1, the Java collections framework does not consider a map a collection. Give a reason for this decision.

**Practice It** Now you can try these exercises at the end of the chapter: R15.1, R15.2, R15.3.

## 15.2 Linked Lists

A **linked list** is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence. In the following sections, you will learn how a linked list manages its elements and how you can use linked lists in your programs.

### 15.2.1 The Structure of Linked Lists

To understand the inefficiency of arrays and the need for a more efficient data structure, imagine a program that maintains a sequence of employee names. If an employee leaves the company, the name must be removed. In an array, the hole in the sequence needs to be closed up by moving all objects that come after it. Conversely, suppose an employee is added in the middle of the sequence. Then all names following the new hire must be moved toward the end. Moving a large number of elements can involve a substantial amount of processing time. A linked list structure avoids this movement.

A linked list uses a sequence of *nodes*. A node is an object that stores an element and references to the neighboring nodes (see Figure 6).

A linked list consists of a number of nodes, each of which has a reference to the next node.



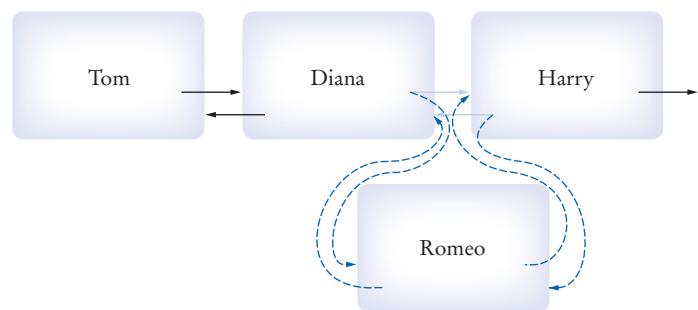
© andrea laurita/Stockphoto.

*Each node in a linked list is connected to the neighboring nodes.*



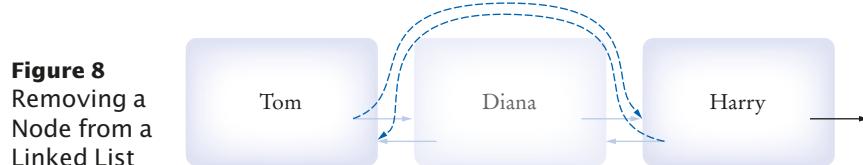
**Figure 6**  
A Linked List

When you insert a new node into a linked list, only the neighboring node references need to be updated (see Figure 7).



**Figure 7** Inserting a Node into a Linked List

The same is true when you remove a node (see Figure 8). What's the catch? Linked lists allow efficient insertion and removal, but element access can be inefficient.



Adding and removing elements at a given location in a linked list is efficient.

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term “random access” is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order. In contrast, sequential access visits the elements in sequence.

Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

### 15.2.2 The LinkedList Class of the Java Collections Framework

The Java library provides a `LinkedList` class in the `java.util` package. It is a **generic class**, just like the `ArrayList` class. That is, you specify the type of the list elements in angle brackets, such as `LinkedList<String>` or `LinkedList<Employee>`.

Table 2 shows important methods of the `LinkedList` class. (Remember that the `LinkedList` class also inherits the methods of the `Collection` interface shown in Table 1.)

As you can see from Table 2, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**. We discuss iterators next.

**Table 2** Working with Linked Lists

|                                                                        |                                                                                                                                                                                 |
|------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>LinkedList&lt;String&gt; list = new LinkedList&lt;&gt;();</code> | An empty list.                                                                                                                                                                  |
| <code>list.addLast("Harry");</code>                                    | Adds an element to the end of the list. Same as <code>add</code> .                                                                                                              |
| <code>list.addFirst("Sally");</code>                                   | Adds an element to the beginning of the list. <code>list</code> is now [Sally, Harry].                                                                                          |
| <code>list.getFirst();</code>                                          | Gets the element stored at the beginning of the list; here "Sally".                                                                                                             |
| <code>list.getLast();</code>                                           | Gets the element stored at the end of the list; here "Harry".                                                                                                                   |
| <code>String removed = list.removeFirst();</code>                      | Removes the first element of the list and returns it. <code>removed</code> is "Sally" and <code>list</code> is [Harry]. Use <code>removeLast</code> to remove the last element. |
| <code>ListIterator&lt;String&gt; iter = list.listIterator()</code>     | Provides an iterator for visiting all list elements (see Table 3 on page 698).                                                                                                  |

### 15.2.3 List Iterators

You use a list iterator to access elements inside a linked list.

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor in a word processor points between two characters (see Figure 9). In the conceptual view, think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

You obtain a list iterator with the `listIterator` method of the `LinkedList` class:

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Note that the iterator class is also a generic type. A `ListIterator<String>` iterates through a list of strings; a `ListIterator<Book>` visits the elements in a `LinkedList<Book>`.

Initially, the iterator points before the first element. You can move the iterator position with the `next` method:

```
iterator.next();
```

The `next` method throws a `NoSuchElementException` if you are already past the end of the list. You should always call the iterator's `hasNext` method before calling `next`—it returns `true` if there is a next element.

```
if (iterator.hasNext())
{
 iterator.next();
}
```

The `next` method returns the element that the iterator is passing. When you use a `ListIterator<String>`, the return type of the `next` method is `String`. In general, the return type of the `next` method matches the list iterator's type parameter (which reflects the type of the elements in the list).

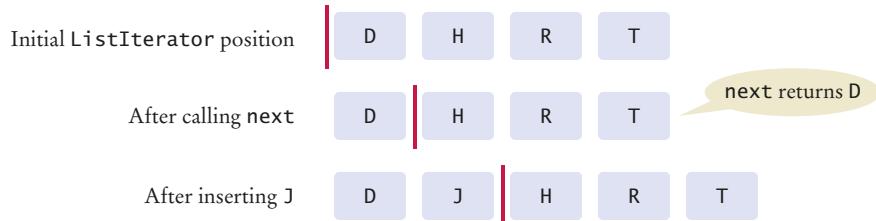
You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
 String name = iterator.next();
 Do something with name.
}
```

As a shorthand, if your loop simply visits all elements of the linked list, you can use the “for each” loop:

```
for (String name : employeeNames)
{
 Do something with name.
}
```

Then you don't have to worry about iterators at all. Behind the scenes, the `for` loop uses an iterator to visit all list elements.



**Figure 9**  
A Conceptual View  
of the List Iterator

**Table 3** Methods of the Iterator and ListIterator Interfaces

|                                                                                     |                                                                                                                                                                                                        |
|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>String s = iter.next();</code>                                                | Assume that <code>iter</code> points to the beginning of the list [Sally] before calling <code>next</code> . After the call, <code>s</code> is "Sally" and the iterator points to the end.             |
| <code>iter.previous();</code><br><code>iter.set("Juliet");</code>                   | The <code>set</code> method updates the last element returned by <code>next</code> or <code>previous</code> . The list is now [Juliet].                                                                |
| <code>iter.hasNext()</code>                                                         | Returns <code>false</code> because the iterator is at the end of the collection.                                                                                                                       |
| <code>if (iter.hasPrevious())</code><br>{<br><code>s = iter.previous();</code><br>} | <code>hasPrevious</code> returns <code>true</code> because the iterator is not at the beginning of the list. <code>previous</code> and <code>hasPrevious</code> are <code>ListIterator</code> methods. |
| <code>iter.add("Diana");</code>                                                     | Adds an element before the iterator position ( <code>ListIterator</code> only). The list is now [Diana, Juliet].                                                                                       |
| <code>iter.next();</code><br><code>iter.remove();</code>                            | <code>remove</code> removes the last element returned by <code>next</code> or <code>previous</code> . The list is now [Diana].                                                                         |

The nodes of the `LinkedList` class store two links: one to the next element and one to the previous one. Such a list is called a **doubly-linked list**. You can use the `previous` and `hasPrevious` methods of the `ListIterator` interface to move the iterator position backward.

The `add` method adds an object before the iterator position.

```
iterator.add("Juliet");
```

You can visualize insertion to be like typing text in a word processor. Each character is inserted after the cursor, then the cursor moves past the inserted character (see Figure 9). Most people never pay much attention to this—you may want to try it out and watch carefully how your word processor inserts characters.

The `remove` method removes the object that was returned by the last call to `next` or `previous`. For example, this loop removes all names that fulfill a certain condition:

```
while (iterator.hasNext())
{
 String name = iterator.next();
 if (condition is fulfilled for name)
 {
 iterator.remove();
 }
}
```

You have to be careful when calling `remove`. It can be called only *once* after calling `next` or `previous`, and you cannot call it immediately after a call to `add`. If you call the method improperly, it throws an `IllegalStateException`.

Table 3 summarizes the methods of the `ListIterator` interface. The `ListIterator` interface extends a more general `Iterator` interface that is suitable for arbitrary collections, not just lists. The table indicates which methods are specific to list iterators.

Following is a sample program that inserts strings into a list and then iterates through the list, adding and removing elements. Finally, the entire list is printed. The comments indicate the iterator position.

**sec02/ListDemo.java**

```

1 import java.util.LinkedList;
2 import java.util.ListIterator;
3
4 /**
5 * This program demonstrates the LinkedList class.
6 */
7 public class ListDemo
8 {
9 public static void main(String[] args)
10 {
11 LinkedList<String> staff = new LinkedList<>();
12 staff.addLast("Diana");
13 staff.addLast("Harry");
14 staff.addLast("Romeo");
15 staff.addLast("Tom");
16
17 // | in the comments indicates the iterator position
18
19 ListIterator<String> iterator = staff.listIterator(); // |DHRT
20 iterator.next(); // D|HRT
21 iterator.next(); // DH|RT
22
23 // Add more elements after second element
24
25 iterator.add("Juliet"); // DHJ|RT
26 iterator.add("Nina"); // DHJN|RT
27
28 iterator.next(); // DHJNR|T
29
30 // Remove last traversed element
31
32 iterator.remove(); // DHJN|T
33
34 // Print all elements
35
36 System.out.println(staff);
37 System.out.println("Expected: [Diana, Harry, Juliet, Nina, Tom]");
38 }
39 }
```

**Program Run**

```
[Diana, Harry, Juliet, Nina, Tom]
Expected: [Diana, Harry, Juliet, Nina, Tom]
```



5. Do linked lists take more storage space than arrays of the same size?
6. Why don't we need iterators with arrays?
7. Suppose the list `letters` contains elements "A", "B", "C", and "D". Draw the contents of the list and the iterator position for the following operations:

```
ListIterator<String> iter = letters.iterator();
iter.next();
iter.next();
iter.remove();
iter.next();
iter.add("E");
```

```

iter.next();
iter.add("F");

```

8. Write a loop that removes all strings with length less than four from a linked list of strings called words.
9. Write a loop that prints every second element of a linked list of strings called words.

**Practice It** Now you can try these exercises at the end of the chapter: R15.5, R15.8, E15.1.



You encounter the benefits of standardization every day. When you buy a light bulb, you can be assured that it fits the socket without having to measure the socket at home and the light bulb in the store. In fact, you may have experienced how painful the lack of standards can be if you have ever purchased a flashlight with nonstandard bulbs. Replacement bulbs for such a flashlight can be difficult and expensive to obtain.

Programmers have a similar desire for standardization. Consider the important goal of platform independence for Java programs. After you compile a Java program into class files, you can execute the class files on any computer that has a Java virtual machine. For this to work, the behavior of the virtual machine has to be strictly defined. If all virtual machines don't behave exactly the same way, then the slogan of "write once, run anywhere" turns into "write once, debug everywhere". In order for multiple implementors to create compatible virtual machines, the virtual machine needed to be *standardized*. That is, someone needed to create a definition of the virtual machine and its expected behavior.

Who creates standards? Some of the most successful standards have been created by volunteer groups such as the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C). The IETF standardizes protocols used in the Internet, such

as the protocol for exchanging e-mail messages. The W3C standardizes the Hypertext Markup Language (HTML), the format for web pages. These standards have been instrumental in the creation of the World Wide Web as an open platform that is not controlled by any one company.

Many programming languages, such as C++ and Scheme, have been standardized by independent standards organizations, such as the American National Standards Institute (ANSI) and the International Organization for Standardization—called ISO for short (not an acronym; see [http://www.iso.org/iso/about/discover-iso\\_isos-name.htm](http://www.iso.org/iso/about/discover-iso_isos-name.htm)). ANSI and ISO are associations of industry professionals who develop standards for everything from car tires to credit card shapes to programming languages.

Many standards are developed by dedicated experts from a multitude of vendors and users, with the objective of creating a set of rules that codifies best practices. But sometimes, standards are very contentious. By 2005, Microsoft started losing government contracts when its customers became concerned that many of their documents were stored in proprietary, undocumented formats. Instead of supporting existing standard formats, or working with an industry group to improve those standards, Microsoft wrote its own standard that simply codified what its product was currently doing, even though that format is widely regarded as being inconsistent and very complex. (The description of the format spans over 6,000 pages.) The company first proposed its standard to the European Computer

Manufacturers Association (ECMA), which approved it with minimal discussion. Then ISO "fast-tracked" it as an existing standard, bypassing the normal technical review mechanism.

For similar reasons, Sun Microsystems, the inventor of Java, never agreed to have a third-party organization standardize the Java language. Instead, they put in place their own standardization process, involving other companies but refusing to relinquish control.

Of course, many important pieces of technology aren't standardized at all. Consider the Windows operating system. Although Windows is often called a de-facto standard, it really is no standard at all. Nobody has ever attempted to define formally what the Windows operating system should do. The behavior changes at the whim of its vendor. That suits Microsoft just fine, because it makes it impossible for a third party to create its own version of Windows.

As a computer professional, there will be many times in your career when you need to make a decision whether to support a particular standard. Consider a simple example. In this chapter, you learn about the collection classes from the standard Java library. However, many computer scientists dislike these classes because of their numerous design issues. Should you use the Java collections in your own code, or should you implement a better set of collections? If you do the former, you have to deal with a design that is less than optimal. If you do the latter, other programmers may have a hard time understanding your code because they aren't familiar with your classes.



© Denis Vorob'yev/iStockphoto.

## 15.3 Sets

As you learned in Section 15.1, a **set** organizes its values in an order that is optimized for efficiency, which may not be the order in which you add elements. Inserting and removing elements is more efficient with a set than with a list.

In the following sections, you will learn how to choose a set implementation and how to work with sets.

### 15.3.1 Choosing a Set Implementation

The HashSet and TreeSet classes both implement the Set interface.

Set implementations arrange the elements so that they can locate them quickly.

You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.

The Set interface in the standard Java library has the same methods as the Collection interface, shown in Table 1. However, there is an essential difference between arbitrary collections and sets. A set does not admit duplicates. If you add an element to a set that is already present, the insertion is ignored.

The HashSet and TreeSet classes implement the Set interface. These two classes provide set implementations based on two different mechanisms, called **hash tables** and **binary search trees**. Both implementations arrange the set elements so that finding, adding, and removing elements is efficient, but they use different strategies.

The basic idea of a hash table is simple. Set elements are grouped into smaller collections of elements that share the same characteristic. You can imagine a hash set of books as having a group for each color, so that books of the same color are in the same group. To find whether a book is already present, you just need to check it against the books in the same color group. Actually, hash tables don't use colors, but integer values (called hash codes) that can be computed from the elements.

In order to use a hash table, the elements must have a method to compute those integer values. This method is called hashCode. The elements must also belong to a class with a properly defined equals method (see Section 9.5.2).

Many classes in the standard library implement these methods, for example String, Integer, Double, Point, Rectangle, Color, and all the collection classes. Therefore, you can form a HashSet<String>, HashSet<Rectangle>, or even a HashSet<HashSet<Integer>>.

Suppose you want to form a set of elements belonging to a class that you declared, such as a HashSet<Book>. Then you need to provide hashCode and equals methods for the class Book. There is one exception to this rule. If all elements are distinct (for example, if your program never has two Book objects with the same author and title), then you can simply inherit the hashCode and equals methods of the Object class.

*On this shelf, books of the same color are grouped together. Similarly, in a hash table, objects with the same hash code are placed in the same group.*



*A tree set keeps its elements in sorted order.*



© Volkan Ersoy/iStockphoto.

The `TreeSet` class uses a different strategy for arranging its elements. Elements are kept in sorted order. For example, a set of books might be arranged by height, or alphabetically by author and title. The elements are not stored in an array—that would make adding and removing elements too inefficient. Instead, they are stored in nodes, as in a linked list. However, the nodes are not arranged in a linear sequence but in a tree shape.

In order to use a `TreeSet`, it must be possible to compare the elements and determine which one is “larger”. You can use a `TreeSet` for classes such as `String` and `Integer` that implement the `Comparable` interface, which we discussed in Section 9.6.3. (That section also shows you how you can implement comparison methods for your own classes.)

As a rule of thumb, you should choose a `TreeSet` if you want to visit the set’s elements in sorted order. Otherwise choose a `HashSet`—as long as the hash function is well chosen, it is a bit more efficient.

When you construct a `HashSet` or `TreeSet`, store the reference in a `Set` variable. For example,

```
Set<String> names = new HashSet<>();
or
Set<String> names = new TreeSet<>();
```

After you construct the collection object, the implementation no longer matters; only the interface is important.

### 15.3.2 Working with Sets

You add and remove set elements with the `add` and `remove` methods:

```
names.add("Romeo");
names.remove("Juliet");
```

As in mathematics, a set collection in Java rejects duplicates. Adding an element has no effect if the element is already in the set. Similarly, attempting to remove an element that isn’t in the set is ignored.

The `contains` method tests whether an element is contained in the set:

```
if (names.contains("Juliet")) . . .
```

The `contains` method uses the `equals` method of the element type. If your set collects `String` or `Integer` objects, you don’t have to worry. Those classes provide an `equals` method. However, if you implemented the element type yourself, then you need to define the `equals` method—see Section 9.5.2.

Finally, to list all elements in the set, get an iterator. As with list iterators, you use the `next` and `hasNext` methods to step through the set.

```
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
```

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

Sets don’t have duplicates. Adding a duplicate of an element that is already present is ignored.

```

String name = iter.next();
Do something with name.
}

```

You can also use the “for each” loop instead of explicitly using an iterator:

```

for (String name : names)
{
 Do something with name.
}

```

A set iterator visits the elements in the order in which the set implementation keeps them.

You cannot add an element to a set at an iterator position.

A set iterator visits the elements in the order in which the set implementation keeps them. This is not necessarily the order in which you inserted them. The order of elements in a hash set seems quite random because the hash code spreads the elements into different groups. When you visit elements of a tree set, they always appear in sorted order, even if you inserted them in a different order.

There is an important difference between the `Iterator` that you obtain from a set and the `ListIterator` that a list yields. The `ListIterator` has an `add` method to add an element at the list iterator position. The `Iterator` interface has no such method. It makes no sense to add an element at a particular position in a set, because the set can order the elements any way it likes. Thus, you always add elements directly to a set, never to an iterator of the set.

However, you can remove a set element at an iterator position, just as you do with list iterators.

Also, the `Iterator` interface has no previous method to go backward through the elements. Because the elements are not ordered, it is not meaningful to distinguish between “going forward” and “going backward”.

**Table 4** Working with Sets

|                                                           |                                                                                                                                       |
|-----------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <code>Set&lt;String&gt; names;</code>                     | Use the interface type for variable declarations.                                                                                     |
| <code>names = new HashSet&lt;&gt;();</code>               | Use a <code>TreeSet</code> if you need to visit the elements in sorted order.                                                         |
| <code>names.add("Romeo");</code>                          | Now <code>names.size()</code> is 1.                                                                                                   |
| <code>names.add("Fred");</code>                           | Now <code>names.size()</code> is 2.                                                                                                   |
| <code>names.add("Romeo");</code>                          | <code>names.size()</code> is still 2. You can't add duplicates.                                                                       |
| <code>if (names.contains("Fred"))</code>                  | The <code>contains</code> method checks whether a value is contained in the set. In this case, the method returns <code>true</code> . |
| <code>System.out.println(names);</code>                   | Prints the set in the format <code>[Fred, Romeo]</code> . The elements need not be shown in the order in which they were inserted.    |
| <code>for (String name : names)</code><br>{<br>. . .<br>} | Use this loop to visit all elements of a set.                                                                                         |
| <code>names.remove("Romeo");</code>                       | Now <code>names.size()</code> is 1.                                                                                                   |
| <code>names.remove("Juliet");</code>                      | It is not an error to remove an element that is not present. The method call has no effect.                                           |

The following program shows a practical application of sets. It reads in all words from a dictionary file that contains correctly spelled words and places them in a set. It then reads all words from a document—here, the book *Alice in Wonderland*—into a second set. Finally, it prints all words from that set that are not in the dictionary set. These are the potential misspellings. (As you can see from the output, we used an American dictionary, and words with British spelling, such as *clamour*, are flagged as potential errors.)

### sec03/SpellCheck.java

```

1 import java.util.HashSet;
2 import java.util.Scanner;
3 import java.util.Set;
4 import java.io.File;
5 import java.io.FileNotFoundException;
6
7 /**
8 * This program checks which words in a file are not present in a dictionary.
9 */
10 public class SpellCheck
11 {
12 public static void main(String[] args)
13 throws FileNotFoundException
14 {
15 // Read the dictionary and the document
16
17 Set<String> dictionaryWords = readWords("words");
18 Set<String> documentWords = readWords("alice30.txt");
19
20 // Print all words that are in the document but not the dictionary
21
22 for (String word : documentWords)
23 {
24 if (!dictionaryWords.contains(word))
25 {
26 System.out.println(word);
27 }
28 }
29 }
30
31 /**
32 * Reads all words from a file.
33 * @param filename the name of the file
34 * @return a set with all lowercased words in the file. Here, a
35 * word is a sequence of upper- and lowercase letters.
36 */
37 public static Set<String> readWords(String filename)
38 throws FileNotFoundException
39 {
40 Set<String> words = new HashSet<>();
41 Scanner in = new Scanner(new File(filename));
42 // Use any characters other than a-z or A-Z as delimiters
43 in.useDelimiter("[^a-zA-Z]+");
44 while (in.hasNext())
45 {
46 words.add(in.next().toLowerCase());
47 }
48 return words;

```

```
49 }
50 }
```

### Program Run

```
neighbouring
croqueted
pennyworth
dutchess
comfits
xii
dinn
clamour
...
```

### SELF CHECK



10. Arrays and lists remember the order in which you added elements; sets do not. Why would you want to use a set instead of an array or list?
11. Why are set iterators different from list iterators?
12. What is wrong with the following test to check whether the `Set<String>` s contains the elements "Tom", "Diana", and "Harry"?
 

```
if (s.toString().equals("[Tom, Diana, Harry]")) . . .
```
13. How can you correctly implement the test of Self Check 12?
14. Write a loop that prints all elements that are in both `Set<String>` s and `Set<String>` t.
15. Suppose you changed line 40 of the SpellCheck program to use a `TreeSet` instead of a `HashSet`. How would the output change?

### Practice It

Now you can try these exercises at the end of the chapter: E15.3, E15.12, E15.13.

### Programming Tip 15.1



### Use Interface References to Manipulate Data Structures

It is considered good style to store a reference to a `HashSet` or `TreeSet` in a variable of type `Set`:

```
Set<String> words = new HashSet<>();
```

This way, you have to change only one line if you decide to use a `TreeSet` instead.

If a method can operate on arbitrary collections, use the `Collection` interface type for the parameter variable:

```
public static void removeLongWords(Collection<String> words)
```

In theory, we should make the same recommendation for the `List` interface, namely to save `ArrayList` and `LinkedList` references in variables of type `List`. However, the `List` interface has get and set methods for random access, even though these methods are very inefficient for linked lists. You can't write efficient code if you don't know whether the methods that you are calling are efficient or not. This is plainly a serious design error in the standard library, and it makes the `List` interface somewhat unattractive.

## 15.4 Maps

The `HashMap` and `TreeMap` classes both implement the `Map` interface.

A **map** allows you to associate elements from a *key set* with elements from a *value collection*. You use a map when you want to look up objects by using a key. For example, Figure 10 shows a map from the names of people to their favorite colors.

Just as there are two kinds of set implementations, the Java library has two implementations for the `Map` interface: `HashMap` and `TreeMap`.

After constructing a `HashMap` or `TreeMap`, you can store the reference to the map object in a `Map` reference:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Use the `put` method to add an association:

```
favoriteColors.put("Juliet", Color.RED);
```

You can change the value of an existing association, simply by calling `put` again:

```
favoriteColors.put("Juliet", Color.BLUE);
```

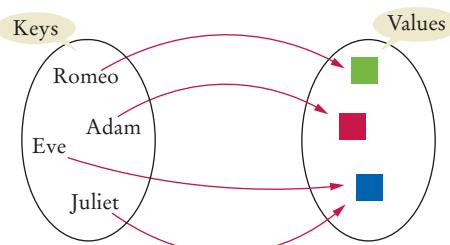
The `get` method returns the value associated with a key.

```
Color juliet'sFavoriteColor = favoriteColors.get("Juliet");
```

If you ask for a key that isn't associated with any values, the `get` method returns `null`.

To remove an association, call the `remove` method with the key:

```
favoriteColors.remove("Juliet");
```



**Figure 10** A Map

**Table 5 Working with Maps**

|                                                                                                                    |                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Map&lt;String, Integer&gt; scores;</code>                                                                    | Keys are strings, values are <code>Integer</code> wrappers. Use the interface type for variable declarations.                                       |
| <code>scores = new TreeMap&lt;&gt;();</code>                                                                       | Use a <code>HashMap</code> if you don't need to visit the keys in sorted order.                                                                     |
| <code>scores.put("Harry", 90);<br/>scores.put("Sally", 95);</code>                                                 | Adds keys and values to the map.                                                                                                                    |
| <code>scores.put("Sally", 100);</code>                                                                             | Modifies the value of an existing key.                                                                                                              |
| <code>int n = scores.get("Sally");<br/>Integer n2 = scores.get("Diana");</code>                                    | Gets the value associated with a key, or <code>null</code> if the key is not present. <code>n</code> is 100, <code>n2</code> is <code>null</code> . |
| <code>System.out.println(scores);</code>                                                                           | Prints <code>scores.toString()</code> , a string of the form {Harry=90, Sally=100}                                                                  |
| <code>for (String key : scores.keySet())<br/>{<br/>    Integer value = scores.get(key);<br/>    . . .<br/>}</code> | Iterates through all map keys and values.                                                                                                           |
| <code>scores.remove("Sally");</code>                                                                               | Removes the key and value.                                                                                                                          |

To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.

Sometimes you want to enumerate all keys in a map. The `keySet` method yields the set of keys. You can then ask the key set for an iterator and get all keys. From each key, you can find the associated value with the `get` method. Thus, the following instructions print all key/value pairs in a map `m` whose keys have type `String`:

```
Set<String> keySet = m.keySet();
for (String key : keySet)
{
 Color value = m.get(key);
 System.out.println(key + " -> " + value);
}
```

This sample program shows a map in action:

### sec04/MapDemo.java

```
1 import java.awt.Color;
2 import java.util.HashMap;
3 import java.util.Map;
4 import java.util.Set;
5
6 /**
7 This program demonstrates a map that maps names to colors.
8 */
9 public class MapDemo
10 {
11 public static void main(String[] args)
12 {
13 Map<String, Color> favoriteColors = new HashMap<>();
14 favoriteColors.put("Juliet", Color.BLUE);
15 favoriteColors.put("Romeo", Color.GREEN);
16 favoriteColors.put("Adam", Color.RED);
17 favoriteColors.put("Eve", Color.BLUE);
18
19 // Print all keys and values in the map
20
21 Set<String> keySet = favoriteColors.keySet();
22 for (String key : keySet)
23 {
24 Color value = favoriteColors.get(key);
25 System.out.println(key + " : " + value);
26 }
27 }
28 }
```

### Program Run

```
Juliet : java.awt.Color[r=0,g=0,b=255]
Adam : java.awt.Color[r=255,g=0,b=0]
Eve : java.awt.Color[r=0,g=0,b=255]
Romeo : java.awt.Color[r=0,g=255,b=0]
```

### SELF CHECK



16. What is the difference between a set and a map?
17. Why is the collection of the keys of a map a set and not a list?
18. Why is the collection of the values of a map not a set?
19. Suppose you want to track how many times each word occurs in a document. Declare a suitable map variable.

**20.** What is a `Map<String, HashSet<String>>`? Give a possible use for such a structure.

**Practice It** Now you can try these exercises at the end of the chapter: R15.20, E15.4, E15.5.

### Java 8 Note 15.1

8

#### Updating Map Entries

Maps are commonly used for counting how often an item occurs. For example, Worked Example 15.1 uses a `Map<String, Integer>` to track how many times a word occurs in a file.

It is a bit tedious to deal with the special case of inserting the first value. Consider the following code from Worked Example 15.1:

```
Integer count = frequencies.get(word); // Get the old frequency count
// If there was none, put 1; otherwise, increment the count
if (count == null) { count = 1; }
else { count = count + 1; }
frequencies.put(word, count);
```

Java 8 adds a useful `merge` method to the `Map` interface. You specify

- A key.
- A value to be used if the key is not yet present.
- A function to compute the updated value if the key is present.

The function is specified as a lambda expression (see Java 8 Note 9.3). For example,

```
frequencies.merge(word, 1, (oldValue, notPresentValue) -> oldValue + notPresentValue);
```

does the same as the four lines of code above. If `word` is not present, the value is set to 1. Otherwise, the old value is incremented.

The `merge` method is also useful if the map values are sets or comma-separated strings—see Exercises E15.6 and E15.7.

### HOW TO 15.1



#### Choosing a Collection

Suppose you need to store objects in a collection. You have now seen a number of different data structures. This How To reviews how to pick an appropriate collection for your application.



© Tom Hahn/  
iStockphoto.

##### Step 1 Determine how you access the values.

You store values in a collection so that you can later retrieve them. How do you want to access individual values? You have several choices:

- Values are accessed by an integer position. Use an `ArrayList`.
- Values are accessed by a key that is not a part of the object. Use a map.
- Values are accessed only at one of the ends. Use a queue (for first-in, first-out access) or a stack (for last-in, first-out access).
- You don't need to access individual values by position. Refine your choice in Steps 3 and 4.

##### Step 2 Determine the element types or key/value types.

For a list or set, determine the type of the elements that you want to store. For example, if you collect a set of books, then the element type is `Book`.

Similarly, for a map, determine the types of the keys and the associated values. If you want to look up books by ID, you can use a `Map<Integer, Book>` or `Map<String, Book>`, depending on your ID type.

### Step 3

Determine whether element or key order matters.

When you visit elements from a collection or keys from a map, do you care about the order in which they are visited? You have several choices:

- Elements or keys must be sorted. Use a `TreeSet` or `TreeMap`. Go to Step 6.
- Elements must be in the same order in which they were inserted. Your choice is now narrowed down to a `LinkedList` or an `ArrayList`.
- It doesn't matter. As long as you get to visit all elements, you don't care in which order. If you chose a map in Step 1, use a `HashMap` and go to Step 5.

### Step 4

For a collection, determine which operations must be efficient.

You have several choices:

- Finding elements must be efficient. Use a `HashSet`.
- It must be efficient to add or remove elements at the beginning, or, provided that you are already inspecting an element there, another position. Use a `LinkedList`.
- You only insert or remove at the end, or you collect so few elements that you aren't concerned about speed. Use an `ArrayList`.

### Step 5

For hash sets and maps, decide whether you need to implement the `hashCode` and `equals` methods.

- If your elements or keys belong to a class that someone else implemented, check whether the class has its own `hashCode` and `equals` methods. If so, you are all set. This is the case for most classes in the standard Java library, such as `String`, `Integer`, `Rectangle`, and so on.
- If not, decide whether you can compare the elements by identity. This is the case if you never construct two distinct elements with the same contents. In that case, you need not do anything—the `hashCode` and `equals` methods of the `Object` class are appropriate.
- Otherwise, you need to implement your own `equals` and `hashCode` methods—see Section 9.5.2 and Special Topic 15.1.

### Step 6

If you use a tree, decide whether to supply a comparator.

Look at the class of the set elements or map keys. Does that class implement the `Comparable` interface? If so, is the sort order given by the `compareTo` method the one you want? If yes, then you don't need to do anything further. This is the case for many classes in the standard library, in particular for `String` and `Integer`.

If not, then your element class must implement the `Comparable` interface (Section 9.6.3), or you must declare a class that implements the `Comparator` interface (see Special Topic 14.4).



## WORKED EXAMPLE 15.1

### Word Frequency



Learn how to create a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file. Go to [wiley.com/go/bjlo2examples](http://wiley.com/go/bjlo2examples) and download Worked Example 15.1.



## Special Topic 15.1

**Hash Functions**

If you use a hash set or hash map with your own classes, you may need to implement a hash function. A **hash function** is a function that computes an integer value, the **hash code**, from an object in such a way that different objects are likely to yield different hash codes. Because hashing is so important, the `Object` class has a `hashCode` method. The call

```
int h = x.hashCode();
```

computes the hash code of any object `x`. If you want to put objects of a given class into a `HashSet` or use the objects as keys in a `HashMap`, the class should override this method. The method should be implemented so that different objects are likely to have different hash codes.

For example, the `String` class declares a hash function for strings that does a good job of producing different integer values for different strings. Table 6 shows some examples of strings and their hash codes.

It is possible for two or more distinct objects to have the same hash code; this is called a *collision*. For example, the strings "Ugh" and "VII" happen to have the same hash code, but these collisions are very rare for strings (see Exercise P15.5).

The `hashCode` method of the `String` class combines the characters of a string into a numerical code. The code isn't simply the sum of the character values—that would not scramble the character values enough. Strings that are permutations of another (such as "eat" and "tea") would all have the same hash code.

Here is the method the standard library uses to compute the hash code for a string:

```
final int HASH_MULTIPLIER = 31;
int h = 0;
for (int i = 0; i < s.length(); i++)
{
 h = HASH_MULTIPLIER * h + s.charAt(i);
}
```

For example, the hash code of "eat" is

$$31 * (31 * 'e' + 'a') + 't' = 100184$$


© one clear vision/iStockphoto.

*A good hash function produces different hash values for each object so that they are scattered about in a hash table.*

A hash function computes an integer value from an object.

A good hash function minimizes *collisions*—identical hash codes for different objects.

**Table 6** Sample Strings and Their Hash Codes

| String   | Hash Code   |
|----------|-------------|
| "eat"    | 100184      |
| "tea"    | 114704      |
| "Juliet" | -2065036585 |
| "Ugh"    | 84982       |
| "VII"    | 84982       |

The hash code of "tea" is quite different, namely

$$31 * (31 * 't' + 'e') + 'a' = 114704$$

(Use the Unicode table from Appendix A to look up the character values: 'a' is 97, 'e' is 101, and 't' is 116.)

For your own classes, you should make up a hash code that combines the hash codes of the instance variables in a similar way. For example, let us declare a hashCode method for the Country class from Section 9.6.1.

There are two instance variables: the country name and the area. First, compute their hash codes. You know how to compute the hash code of a string. To compute the hash code of a floating-point number, first wrap the floating-point number into a Double object, and then compute its hash code.

Override hashCode methods in your own classes by combining the hash codes for the instance variables.

```
public class Country
{
 ...
 public int hashCode()
 {
 int h1 = name.hashCode();
 int h2 = new Double(area).hashCode();
 ...
 }
}
```

Then combine the two hash codes:

```
final int HASH_MULTIPLIER = 31;
int h = HASH_MULTIPLIER * h1 + h2;
return h;
```

However, it is easier to use the Objects.hash method which takes the hash codes of all arguments and combines them with a multiplier.

```
public int hashCode()
{
 return Objects.hash(name, area);
}
```

When you supply your own hashCode method for a class, you must also provide a compatible equals method. The equals method is used to differentiate between two objects that happen to have the same hash code.

The equals and hashCode methods must be *compatible* with each other. Two objects that are equal must yield the same hash code.

A class's hashCode method must be compatible with its equals method.

You get into trouble if your class declares an equals method but not a hashCode method. Suppose the Country class declares an equals method (checking that the name and area are the same), but no hashCode method. Then the hashCode method is inherited from the Object superclass. That method computes a hash code from the *memory location* of the object. Then it is very likely that two objects with the same contents will have different hash codes, in which case a hash set will store them as two distinct objects.

However, if you declare *neither* equals *nor* hashCode, then there is no problem. The equals method of the Object class considers two objects equal only if their memory location is the same. That is, the Object class has compatible equals and hashCode methods. Of course, then the notion of equality is very restricted: Only identical objects are considered equal. That can be a perfectly valid notion of equality, depending on your application.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj1o2code](http://wiley.com/go/bj1o2code) to download a program that demonstrates a hash set with objects of the Country class.

## 15.5 Stacks, Queues, and Priority Queues

In the following sections, we cover stacks, queues, and priority queues. These data structures each have a different policy for data removal. Removing an element yields the most recently added element, the least recently added, or the element with the highest priority.

### 15.5.1 Stacks

A stack is a collection of elements with “last-in, first-out” retrieval.

© budgetstockphoto/iStockphoto.



The Undo key pops commands off a stack so that the last command is the first to be undone.

A **stack** lets you insert and remove elements only at one end, traditionally called the *top* of the stack. New items can be added to the top of the stack. Items are removed from the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last-in, first-out* or *LIFO* order. For example, if you add items A, B, and C and then remove them, you obtain C, B, and A. With stacks, the addition and removal operations are called *push* and *pop*.

```
Stack<String> s = new Stack<>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
 System.out.print(s.pop() + " ");
} // Prints C B A
```

There are many applications for stacks in computer science. Consider the undo feature of a word processor. It keeps the issued commands in a stack. When you select “Undo”, the *last* command is undone, then the next-to-last, and so on.

Another important example is the **run-time stack** that a processor or virtual machine keeps to store the values of variables in nested methods. Whenever a new method is called, its parameter variables and local variables are pushed onto a stack. When the method exits, they are popped off again.

You will see other applications in Section 15.6.

The Java library provides a simple `Stack` class with methods `push`, `pop`, and `peek`—the latter gets the top element of the stack but does not remove it (see Table 7).



The last pancake that has been added to this stack will be the first one that is consumed.

© John Madden/iStockphoto.

**Table 7** Working with Stacks

|                                                                               |                                                                                                                                                                                          |
|-------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Stack&lt;Integer&gt; s = new Stack&lt;&gt;();</code>                    | Constructs an empty stack.                                                                                                                                                               |
| <code>s.push(1);</code><br><code>s.push(2);</code><br><code>s.push(3);</code> | Adds to the top of the stack; <code>s</code> is now [1, 2, 3].<br>(Following the <code>toString</code> method of the <code>Stack</code> class, we show the top of the stack at the end.) |
| <code>int top = s.pop();</code>                                               | Removes the top of the stack; <code>top</code> is set to 3 and <code>s</code> is now [1, 2].                                                                                             |
| <code>head = s.peek();</code>                                                 | Gets the top of the stack without removing it;<br><code>head</code> is set to 2.                                                                                                         |

## 15.5.2 Queues

A queue is a collection of elements with “first-in, first-out” retrieval.

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*). Queues yield items in a *first-in, first-out* or *FIFO* fashion. Items are removed in the same order in which they were added.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue. When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the “first-in, first-out” rule, which is a fair arrangement for users of the shared printer.

The `Queue` interface in the standard Java library has methods `add` to add an element to the tail of the queue, `remove` to remove the head of the queue, and `peek` to get the head element of the queue without removing it (see Table 8).

The `LinkedList` class implements the `Queue` interface. Whenever you need a queue, simply initialize a `Queue` variable with a `LinkedList` object:

```
Queue<String> q = new LinkedList<>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " ");}
// Prints A B C
```

The standard library provides several queue classes that we do not discuss in this book. Those classes are intended for work sharing when multiple activities (called threads) run in parallel.



© paul kline/iStockphoto.



*When you retrieve an item from a priority queue, you always get the most urgent one.*



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to download programs that demonstrate stacks, queues, and priority queues.

It is customary to give low values to urgent priorities, with priority 1 denoting the most urgent priority. Thus, each removal operation extracts the *minimum* element from the queue.

For example, consider this code in which we add objects of a class `WorkOrder` into a priority queue. Each work order has a priority and a description.

```
PriorityQueue<WorkOrder> q = new PriorityQueue<>();
q.add(new WorkOrder(3, "Shampoo carpets"));
q.add(new WorkOrder(1, "Fix broken sink"));
q.add(new WorkOrder(2, "Order cleaning supplies"));
```

When calling `q.remove()` for the first time, the work order with priority 1 is removed. The next call to `q.remove()` removes the work order whose priority is highest among those remaining in the queue—in our example, the work order with priority 2. If there happen to be two elements with the same priority, the priority queue will break ties arbitrarily.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. (See Section 9.6.3 for a description of that interface type.)

Table 9 shows the methods of the `PriorityQueue` class in the standard Java library.

**Table 9** Working with Priority Queues

|                                                                               |                                                                                                                          |
|-------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>PriorityQueue&lt;Integer&gt; q = new PriorityQueue&lt;&gt;();</code>    | This priority queue holds <code>Integer</code> objects. In practice, you would use objects that describe tasks.          |
| <code>q.add(3); q.add(1); q.add(2);</code>                                    | Adds values to the priority queue.                                                                                       |
| <code>int first = q.remove();</code><br><code>int second = q.remove();</code> | Each call to <code>remove</code> removes the most urgent item: <code>first</code> is set to 1, <code>second</code> to 2. |
| <code>int next = q.peek();</code>                                             | Gets the smallest value in the priority queue without removing it.                                                       |

#### SELF CHECK



21. Why would you want to declare a variable as  
`Queue<String> q = new LinkedList<>();`  
 instead of simply declaring it as a linked list?
22. Why wouldn't you want to use an array list for implementing a queue?
23. What does this code print?  

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) { System.out.print(q.remove() + " "); }
```
24. Why wouldn't you want to use a stack to manage print jobs?
25. In the sample code for a priority queue, we used a `WorkOrder` class. Could we have used strings instead?

```
PriorityQueue<String> q = new PriorityQueue<>();
q.add("3 - Shampoo carpets");
q.add("1 - Fix broken sink");
q.add("2 - Order cleaning supplies");
```

**Practice It** Now you can try these exercises at the end of the chapter: R15.15, E15.8, E15.9.

## 15.6 Stack and Queue Applications

Stacks and queues are, despite their simplicity, very versatile data structures. In the following sections, you will see some of their most useful applications.

### 15.6.1 Balancing Parentheses

A stack can be used to check whether parentheses in an expression are balanced.

In Common Error 2.5, you saw a simple trick for detecting unbalanced parentheses in an expression such as

$$-(b * b - (4 * a * c) ) / (2 * a)$$

|   |   |    |   |   |
|---|---|----|---|---|
| 1 | 2 | 10 | 1 | 0 |
|---|---|----|---|---|

Increment a counter when you see a ( and decrement it when you see a ). The counter should never be negative, and it should be zero at the end of the expression.

That works for expressions in Java, but in mathematical notation, one can have more than one kind of parentheses, such as

$$-\{ [b \cdot b - (4 \cdot a \cdot c)] / (2 \cdot a) \}$$

To see whether such an expression is correctly formed, place the parentheses on a stack:

**When you see an opening parenthesis, push it on the stack.**

**When you see a closing parenthesis, pop the stack.**

**If the opening and closing parentheses don't match**

**The parentheses are unbalanced. Exit.**

**If at the end the stack is empty**

**The parentheses are balanced.**

**Else**

**The parentheses are not balanced.**



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bjlo2code](http://wiley.com/go/bjlo2code) to download a program for checking balanced parentheses.

Here is a walkthrough of the sample expression:

| Stack               | Unread expression                        | Comments                     |
|---------------------|------------------------------------------|------------------------------|
| Empty               | $-\{ [b * b - (4 * a * c)] / (2 * a) \}$ |                              |
| {                   | $[b * b - (4 * a * c)] / (2 * a) \}$     |                              |
| { [                 | $b * b - (4 * a * c)] / (2 * a) \}$      |                              |
| { [ (               | $4 * a * c)] / (2 * a) \}$               |                              |
| { [ ( ] / (2 * a) } |                                          | ( matches )                  |
| { [ ( ] / (2 * a) } |                                          | [ matches ]                  |
| { [ ( ] 2 * a) }    |                                          |                              |
| { [ ( ] }           |                                          | ( matches )                  |
| Empty               | No more input                            | { matches }                  |
|                     |                                          | The parentheses are balanced |

## 15.6.2 Evaluating Reverse Polish Expressions

Use a stack to evaluate expressions in reverse Polish notation.

Consider how you write arithmetic expressions, such as  $(3 + 4) \times 5$ . The parentheses are needed so that 3 and 4 are added before multiplying the result by 5.

However, you can eliminate the parentheses if you write the operators *after* the numbers, like this: 3 4 + 5 × (see Special Topic 15.2 on page 723). To evaluate this expression, apply + to 3 and 4, yielding 7, and then simplify 7 5 × to 35. It gets trickier for complex expressions. For example, 3 4 5 + × means to compute 4 5 + (that is, 9), and then evaluate 3 9 ×. If we evaluate this expression left-to-right, we need to leave the 3 somewhere while we work on 4 5 +. Where? We put it on a stack. The algorithm for evaluating reverse Polish expressions is simple:

```
If you read a number
 Push it on the stack.
Else if you read an operand
 Pop two values off the stack.
 Combine the values with the operand.
 Push the result back onto the stack.
Else if there is no more input
 Pop and display the result.
```

Here is a walkthrough of evaluating the expression 3 4 5 + ×:

| Stack | Unread expression | Comments                        |
|-------|-------------------|---------------------------------|
| Empty | 3 4 5 + ×         |                                 |
| 3     | 4 5 + ×           | Numbers are pushed on the stack |
| 3 4   | 5 + ×             |                                 |
| 3 4 5 | + ×               |                                 |
| 3 9   | x                 | Pop 4 and 5, push 4 5 +         |
| 27    | No more input     | Pop 3 and 9, push 3 9 ×         |
| Empty |                   | Pop and display the result, 27  |

The following program simulates a reverse Polish calculator:

### sec06\_02/Calculator.java

```

1 import java.util.Scanner;
2 import java.util.Stack;
3
4 /**
5 * This calculator uses the reverse Polish notation.
6 */
7 public class Calculator
8 {
9 public static void main(String[] args)
10 {
11 Scanner in = new Scanner(System.in);
12 Stack<Integer> results = new Stack<>();
13 System.out.println("Enter one number or operator per line, Q to quit. ");
14 boolean done = false;
```

```

15 while (!done)
16 {
17 String input = in.nextLine();
18
19 // If the command is an operator, pop the arguments and push the result
20
21 if (input.equals("+"))
22 {
23 results.push(results.pop() + results.pop());
24 }
25 else if (input.equals("-"))
26 {
27 Integer arg2 = results.pop();
28 results.push(results.pop() - arg2);
29 }
30 else if (input.equals("*") || input.equals("x"))
31 {
32 results.push(results.pop() * results.pop());
33 }
34 else if (input.equals("/"))
35 {
36 Integer arg2 = results.pop();
37 results.push(results.pop() / arg2);
38 }
39 else if (input.equals("Q") || input.equals("q"))
40 {
41 done = true;
42 }
43 else
44 {
45 // Not an operator--push the input value
46
47 results.push(Integer.parseInt(input));
48 }
49 System.out.println(results);
50 }
51 }
52 }
```

### 15.6.3 Evaluating Algebraic Expressions

Using two stacks,  
you can evaluate  
expressions in  
standard algebraic  
notation.

In the preceding section, you saw how to evaluate expressions in reverse Polish notation, using a single stack. If you haven't found that notation attractive, you will be glad to know that one can evaluate an expression in the standard algebraic notation using two stacks—one for numbers and one for operators.

*Use two stacks to evaluate algebraic expressions.*



© Jorge Delgado/Stockphoto

First, consider a simple example, the expression  $3 + 4$ . We push the numbers on the number stack and the operators on the operator stack. Then we pop both numbers and the operator, combine the numbers with the operator, and push the result.

|   | Number stack | Operator stack | Unprocessed input | Comments          |
|---|--------------|----------------|-------------------|-------------------|
|   | Empty        | Empty          | $3 + 4$           |                   |
| 1 | 3            |                | + 4               |                   |
| 2 | 3            | +              | 4                 |                   |
| 3 | 4<br>3       | +              | No more input     | Evaluate the top. |
| 4 | 7            |                |                   | The result is 7.  |

This operation is fundamental to the algorithm. We call it “evaluating the top”.

In algebraic notation, each operator has a *precedence*. The + and - operators have the lowest precedence, \* and / have a higher (and equal) precedence.

Consider the expression  $3 \times 4 + 5$ . Here are the first processing steps:

|   | Number stack | Operator stack | Unprocessed input | Comments             |
|---|--------------|----------------|-------------------|----------------------|
|   | Empty        | Empty          | $3 \times 4 + 5$  |                      |
| 1 | 3            |                | $\times 4 + 5$    |                      |
| 2 | 3            | *              | $4 + 5$           |                      |
| 3 | 4<br>3       | *              | + 5               | Evaluate × before +. |

Because  $\times$  has a higher precedence than  $+$ , we are ready to evaluate the top:

|   | Number stack | Operator stack | Comments            |
|---|--------------|----------------|---------------------|
| 4 | 12           | +              | 5                   |
| 5 | 5<br>12      | +              | No more input       |
| 6 | 17           |                | That is the result. |

With the expression,  $3 + 4 \times 5$ , we add  $\times$  to the operator stack because we must first read the next number; then we can evaluate  $\times$  and then the  $+$ :

|   | Number stack | Operator stack | Unprocessed input | Comments |
|---|--------------|----------------|-------------------|----------|
|   | Empty        | Empty          | $3 + 4 \times 5$  |          |
| 1 | 3            |                | $+ 4 \times 5$    |          |
| 2 | 3            | +              | $4 \times 5$      |          |

|   |        |               |            |                       |
|---|--------|---------------|------------|-----------------------|
| 3 | 4<br>3 | +<br><br><br> | $\times 5$ | Don't evaluate + yet. |
| 4 | 4<br>3 | $\times$<br>+ | 5          |                       |

In other words, we keep operators on the stack until they are ready to be evaluated. Here is the remainder of the computation:

|   | Number stack | Operator stack | Comments      |                     |
|---|--------------|----------------|---------------|---------------------|
| 5 | 5<br>4<br>3  | $\times$<br>+  | No more input | Evaluate the top.   |
| 6 | 20<br>3      | +              |               | Evaluate top again. |
| 7 | 23           |                |               | That is the result. |

To see how parentheses are handled, consider the expression  $3 \times (4 + 5)$ . A ( is pushed on the operator stack. The + is pushed as well. When we encounter the ), we know that we are ready to evaluate the top until the matching ( reappears:

|   | Number stack | Operator stack | Unprocessed input  | Comments                     |
|---|--------------|----------------|--------------------|------------------------------|
|   | Empty        | Empty          | $3 \times (4 + 5)$ |                              |
| 1 | 3            |                | $\times (4 + 5)$   |                              |
| 2 | 3            | $\times$       | (4 + 5)            |                              |
| 3 | 3            | (              | 4 + 5)             | Don't evaluate $\times$ yet. |
| 4 | 4<br>3       | (              | + 5)               |                              |
| 5 | 4<br>3       | +              | 5)                 |                              |
| 6 | 5<br>4<br>3  | +              | )                  | Evaluate the top.            |
| 7 | 9<br>3       | (              | No more input      | Pop (.                       |
| 8 | 9<br>3       | ×              |                    | Evaluate top again.          |
| 9 | 27           |                |                    | That is the result.          |

Here is the algorithm:

```

If you read a number
 Push it on the number stack.
Else if you read a (
 Push it on the operator stack.
Else if you read an operator op
 While the top of the stack has a higher precedence than op
 Evaluate the top.
 Push op on the operator stack.
Else if you read a)
 While the top of the stack is not a (
 Evaluate the top.
 Pop the (.
Else if there is no more input
 While the operator stack is not empty
 Evaluate the top.

```

At the end, the remaining value on the number stack is the value of the expression.

The algorithm makes use of this helper method that evaluates the topmost operator with the topmost numbers:

#### Evaluate the top:

```

Pop two numbers off the number stack.
Pop an operator off the operator stack.
Combine the numbers with that operator.
Push the result on the number stack.

```

#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj102code](http://wiley.com/go/bj102code) to get the complete code for the expression calculator.

Use a stack to remember choices you haven't yet made so that you can backtrack to them.

## 15.6.4 Backtracking

Suppose you are inside a maze. You need to find the exit. What should you do when you come to an intersection? You can continue exploring one of the paths, but you will want to remember the other ones. If your chosen path didn't work, you can go back to one of the other choices and try again.

Of course, as you go along one path, you may reach further intersections, and you need to remember your choice again. Simply use a stack to remember the paths that still need to be tried. The process of returning to a choice point and trying another choice is called *backtracking*. By using a stack, you return to your more recent choices before you explore the earlier ones.

Figure 11 shows an example. We start at a point in the maze, at position (3, 4). There are four possible paths. We push them all on a stack ❶. We pop off the topmost one, traveling north from (3, 4). Following this path leads to position (1, 4). We now push two choices on the stack, going west or east ❷. Both of them lead to dead ends ❸ ❹.

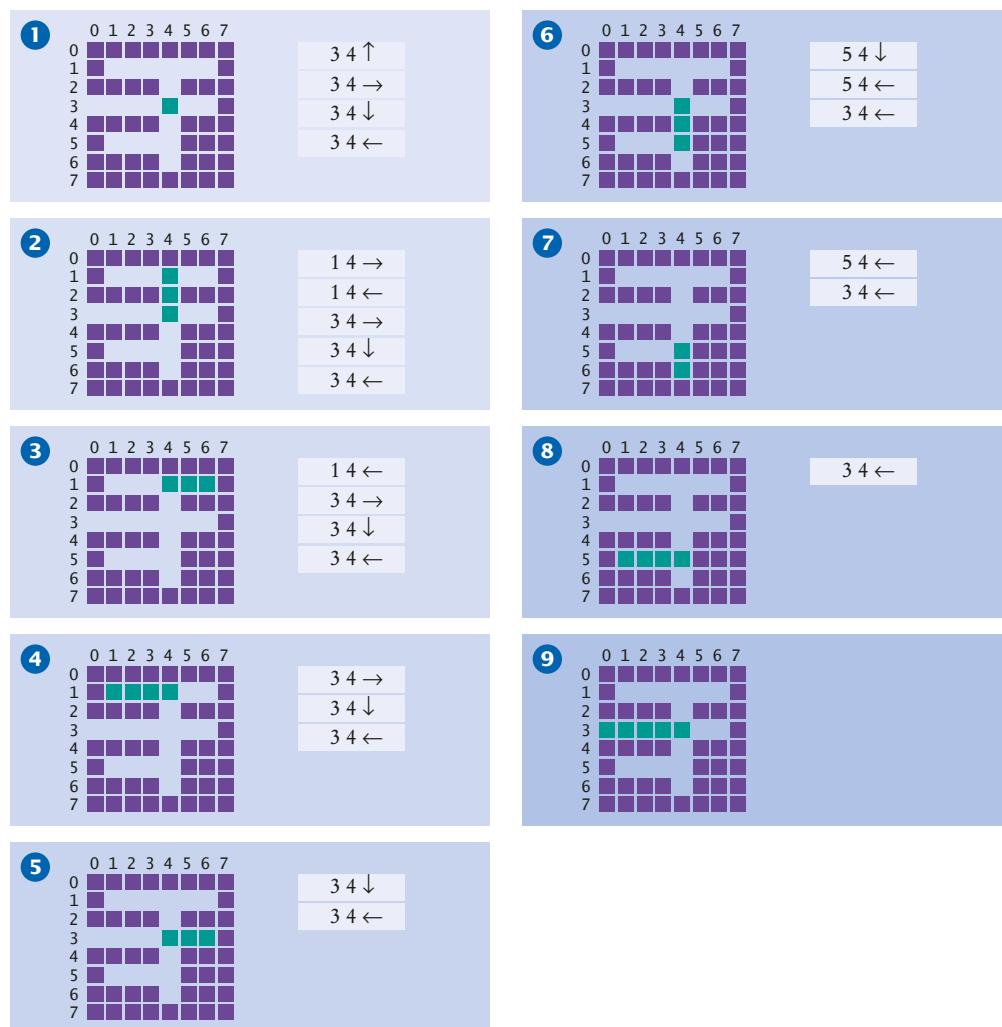
Now we pop off the path from (3, 4) going east. That too is a dead end ❺. Next is the path from (3, 4) going south. At (5, 4), it comes to an intersection. Both choices are pushed on the stack ❻. They both lead to dead ends ❼ ❼.

Finally, the path from (3, 4) going west leads to an exit ❾.



A stack can be used to track positions in a maze.

© Skip O'Donnell/  
iStockphoto.



**Figure 11** Backtracking Through a Maze

Using a stack, we have found a path out of the maze. Here is the pseudocode for our maze-finding algorithm:

```

Push all paths from the point on which you are standing on a stack.
While the stack is not empty
 Pop a path from the stack.
 Follow the path until you reach an exit, intersection, or dead end.
 If you found an exit
 Congratulations!
 Else if you found an intersection
 Push all paths meeting at the intersection, except the current one, onto the stack.

```

This algorithm will find an exit from the maze, provided that the maze has no *cycles*. If it is possible that you can make a circle and return to a previously visited intersection along a different sequence of paths, then you need to work harder—see Exercise E15.21.

How you implement this algorithm depends on the description of the maze. In the example code, we use a two-dimensional array of characters, with spaces for corridors and asterisks for walls, like this:

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

In the example code, a `Path` object is constructed with a starting position and a direction (North, East, South, or West). The `Maze` class has a method that extends a path until it reaches an intersection or exit, or until it is blocked by a wall, and a method that computes all paths from an intersection point.

Note that you can use a queue instead of a stack in this algorithm. Then you explore the earlier alternatives before the later ones. This can work just as well for finding an answer, but it isn't very intuitive in the context of exploring a maze—you would have to imagine being teleported back to the initial intersections rather than just walking back to the last one.



#### FULL CODE EXAMPLE

Go to [wiley.com/go/bj1o2code](http://wiley.com/go/bj1o2code) to download a complete program demonstrating backtracking.

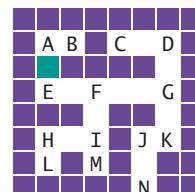


#### SELF CHECK

26. What is the value of the reverse Polish notation expression  $2\ 3\ 4\ 5\ \times\ +\ \times$ ?
27. Why does the branch for the subtraction operator in the `calculator` program not simply execute  
`results.push(results.pop() - results.pop());`
28. In the evaluation of the expression  $3 - 4 + 5$  using the algorithm of Section 15.6.3, which operator gets evaluated first?
29. In the algorithm of Section 15.6.3, are the operators on the operator stack always in increasing precedence?
30. Consider the simple maze at right. Assuming that we start at the marked point and push paths in the order West, South, East, North, in which order are the lettered points visited, using the algorithm of Section 15.6.4?

#### Practice It

Now you can try these exercises at the end of the chapter: R15.25, E15.18, E15.20, E15.21, E15.22.



#### WORKED EXAMPLE 15.2



#### Simulating a Queue of Waiting Customers

Learn how to use a queue to simulate an actual queue of waiting customers. Go to [wiley.com/go/bj1o2examples](http://wiley.com/go/bj1o2examples) and download Worked Example 15.2.



Photodisc/Punchstock.



#### VIDEO EXAMPLE 15.1

#### Building a Table of Contents

In this Video Example, you will see how to build a table of contents for a book. Go to [wiley.com/go/bj1o2videos](http://wiley.com/go/bj1o2videos) to view Video Example 15.1.

## Special Topic 15.2

**Reverse Polish Notation**

In the 1920s, the Polish mathematician Jan Łukasiewicz realized that it is possible to dispense with parentheses in arithmetic expressions, provided that you write the operators *before* their arguments, for example,  $+ 3 4$  instead of  $3 + 4$ . Thirty years later, Australian computer scientist Charles Hamblin noted that an even better scheme would be to have the operators *follow* the operands. This was termed **reverse Polish notation** or RPN.

| Standard Notation        | Reverse Polish Notation |
|--------------------------|-------------------------|
| $3 + 4$                  | $3 4 +$                 |
| $3 + 4 \times 5$         | $3 4 5 \times +$        |
| $3 \times (4 + 5)$       | $3 4 5 + \times$        |
| $(3 + 4) \times (5 + 6)$ | $3 4 + 5 6 + \times$    |
| $3 + 4 + 5$              | $3 4 + 5 +$             |

Reverse Polish notation might look strange to you, but that is just an accident of history. Had earlier mathematicians realized its advantages, today's schoolchildren might be using it and not worrying about precedence rules and parentheses.

In 1972, Hewlett-Packard introduced the HP 35 calculator that used reverse Polish notation. The calculator had no keys labeled with parentheses or an equals symbol. There is just a key labeled ENTER to push a number onto a stack. For that reason, Hewlett-Packard's marketing department used to refer to their product as "the calculators that have no equal".

Over time, calculator vendors have adapted to the standard algebraic notation rather than forcing its users to learn a new notation. However, those users who have made the effort to learn reverse Polish notation tend to be fanatic proponents, and to this day, some Hewlett-Packard calculator models still support it.



Courtesy of Nigel Tout.

*The Calculator with No Equal*

## CHAPTER SUMMARY

### Understand the architecture of the Java collections framework.



- A collection groups together elements and allows them to be retrieved later.
- A list is a collection that remembers the order of its elements.
- A set is an unordered collection of unique elements.
- A map keeps associations between key and value objects.

### Understand and use linked lists.



- A linked list consists of a number of nodes, each of which has a reference to the next node.
- Adding and removing elements at a given position in a linked list is efficient.
- Visiting the elements of a linked list in sequential order is efficient, but random access is not.
- You use a list iterator to access elements inside a linked list.

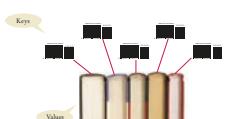
### Choose a set implementation and use it to manage sets of values.



- The HashSet and TreeSet classes both implement the Set interface.
- Set implementations arrange the elements so that they can locate them quickly.
- You can form hash sets holding objects of type String, Integer, Double, Point, Rectangle, or Color.
- You can form tree sets for any class that implements the Comparable interface, such as String or Integer.
- Sets don't have duplicates. Adding a duplicate of an element that is already present is ignored.
- A set iterator visits the elements in the order in which the set implementation keeps them.
- You cannot add an element to a set at an iterator position.



### Use maps to model associations between keys and values.



- The HashMap and TreeMap classes both implement the Map interface.
- To find all keys and values in a map, iterate through the key set and find the values that correspond to the keys.
- A hash function computes an integer value from an object.
- A good hash function minimizes *collisions*—identical hash codes for different objects.
- Override hashCode methods in your own classes by combining the hash codes for the instance variables.
- A class's hashCode method must be compatible with its equals method.



### Use the Java classes for stacks, queues, and priority queues.



- A stack is a collection of elements with “last-in, first-out” retrieval.
- A queue is a collection of elements with “first-in, first-out” retrieval.
- When removing an element from a priority queue, the element with the most urgent priority is retrieved.



### Solve programming problems using stacks and queues.



- A stack can be used to check whether parentheses in an expression are balanced.
- Use a stack to evaluate expressions in reverse Polish notation.
- Using two stacks, you can evaluate expressions in standard algebraic notation.
- Use a stack to remember choices you haven’t yet made so that you can backtrack to them.

### STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

`java.util.Collection<E>`  
`add`  
`contains`  
`iterator`  
`remove`  
`size`  
`java.util.HashMap<K, V>`  
`java.util.HashSet<E>`  
`java.util.Iterator<E>`  
`hasNext`  
`next`  
`remove`  
`java.util.LinkedList<E>`  
`addFirst`  
`addLast`  
`getFirst`  
`getLast`  
`removeFirst`  
`removeLast`

`java.util.List<E>`  
`listIterator`  
`java.util.ListIterator<E>`  
`add`  
`hasPrevious`  
`previous`  
`set`  
`java.util.Map<K, V>`  
`get`  
`keySet`  
`put`  
`remove`  
`java.util.Objects`  
`hash`

`java.util.PriorityQueue<E>`  
`remove`  
`java.util.Queue<E>`  
`peek`  
`java.util.Set<E>`  
`java.util.Stack<E>`  
`peek`  
`pop`  
`push`  
`java.util.TreeMap<K, V>`  
`java.util.TreeSet<K, V>`

### REVIEW EXERCISES

- R15.1** An invoice contains a collection of purchased items. Should that collection be implemented as a list or set? Explain your answer.
- R15.2** Consider a program that manages an appointment calendar. Should it place the appointments into a list, stack, queue, or priority queue? Explain your answer.
- R15.3** One way of implementing a calendar is as a map from date objects to event objects. However, that only works if there is a single event for a given date. How can you use another collection type to allow for multiple events on a given date?

■ ■ **R15.4** Look up the descriptions of the methods `addAll`, `removeAll`, `retainAll`, and `containsAll` in the `Collection` interface. Describe how these methods can be used to implement common operations on sets (union, intersection, difference, subset).

■ **R15.5** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
System.out.println(staff.removeFirst());
```

■ **R15.6** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addFirst("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.7** Explain what the following code prints. Draw a picture of the linked list after each step.

```
LinkedList<String> staff = new LinkedList<>();
staff.addFirst("Harry");
staff.addLast("Diana");
staff.addFirst("Tom");
System.out.println(staff.removeLast());
System.out.println(staff.removeFirst());
System.out.println(staff.removeLast());
```

■ **R15.8** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
if (iterator.next().equals("Tom")) { iterator.remove(); }
while (iterator.hasNext()) { System.out.println(iterator.next()); }
```

■ **R15.9** Explain what the following code prints. Draw a picture of the linked list and the iterator position after each step.

```
LinkedList<String> staff = new LinkedList<>();
ListIterator<String> iterator = staff.listIterator();
iterator.add("Tom");
iterator.add("Diana");
iterator.add("Harry");
iterator = staff.listIterator();
iterator.next();
iterator.next();
```

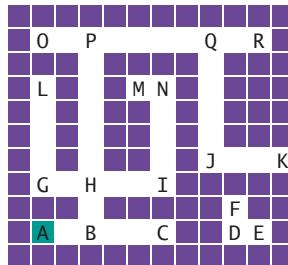
```

iterator.add("Romeo");
iterator.next();
iterator.add("Juliet");
iterator = staff.listIterator();
iterator.next();
iterator.remove();
while (iterator.hasNext()) { System.out.println(iterator.next()); }

```

- R15.10** You are given a linked list of strings. How do you remove all elements with length less than or equal to three?
- R15.11** Repeat Exercise R15.10, using the `removeIf` method. (Read the description in the API of the `Collection` interface.) Use a lambda expression (see Java 8 Note 9.3).
- R15.12** What advantages do linked lists have over arrays? What disadvantages do they have?
- R15.13** Suppose you need to organize a collection of telephone numbers for a company division. There are currently about 6,000 employees, and you know that the phone switch can handle at most 10,000 phone numbers. You expect several hundred look-ups against the collection every day. Would you use an array list or a linked list to store the information?
- R15.14** Suppose you need to keep a collection of appointments. Would you use a linked list or an array list of `Appointment` objects?
- R15.15** Suppose you write a program that models a card deck. Cards are taken from the top of the deck and given out to players. As cards are returned to the deck, they are placed on the bottom of the deck. Would you store the cards in a stack or a queue?
- R15.16** Suppose the strings "A" . . . "Z" are pushed onto a stack. Then they are popped off the stack and pushed onto a second stack. Finally, they are all popped off the second stack and printed. In which order are the strings printed?
- R15.17** What is the difference between a set and a map?
- R15.18** The union of two sets  $A$  and  $B$  is the set of all elements that are contained in  $A$ ,  $B$ , or both. The intersection is the set of all elements that are contained in  $A$  and  $B$ . How can you compute the union and intersection of two sets, using the `add` and `contains` methods, together with an iterator?
- R15.19** How can you compute the union and intersection of two sets, using some of the methods that the `java.util.Set` interface provides, but without using an iterator? (Look up the interface in the API documentation.)
- R15.20** Can a map have two keys with the same value? Two values with the same key?
- R15.21** A map can be implemented as a set of  $(key, value)$  pairs. Explain.
- R15.22** How can you print all key/value pairs of a map, using the `keySet` method? The `entrySet` method? The `forEach` method with a lambda expression? (See Java 8 Note 9.3 on lambda expressions.)
- R15.23** Verify the hash code of the string "Juliet" in Table 6.
- R15.24** Verify that the strings "VII" and "Ugh" have the same hash code.

- **R15.25** Consider the algorithm for traversing a maze from Section 15.6.4 Assume that we start at position A and push in the order West, South, East, and North. In which order will the lettered locations of the sample maze be visited?



- **R15.26** Repeat Exercise R15.25, using a queue instead of a stack.

### PRACTICE EXERCISES

- **E15.1** Write a method

```
public static void downsize(LinkedList<String> employeeNames, int n)
```

that removes every  $n$ th employee from a linked list.

- **E15.2** Write a method

```
public static void reverse(LinkedList<String> strings)
```

that reverses the entries in a linked list.

- **E15.3** Implement the *sieve of Eratosthenes*: a method for computing prime numbers, known to the ancient Greeks. This method computes all prime numbers up to  $n$ . Choose an  $n$ . First insert all numbers from 2 to  $n$  into a set. Then erase all multiples of 2 (except 2); that is, 4, 6, 8, 10, 12, .... Erase all multiples of 3; that is, 6, 9, 12, 15, .... Go up to  $\sqrt{n}$ . Then print the set.



© martin meelligott/Stockphoto.

- **E15.4** Write a program that keeps a map in which both keys and values are strings—the names of students and their course grades. Prompt the user of the program to add or remove students, to modify grades, or to print all grades. The printout should be sorted by name and formatted like this:

```
Carl: B+
Joe: C
Sarah: A
```

- **E15.5** Write a program that reads a Java source file and produces an index of all identifiers in the file. For each identifier, print all lines in which it occurs. For simplicity, we will consider each string consisting only of letters, numbers, and underscores an identifier. Declare a Scanner `in` for reading from the source file and call

```
in.useDelimiter("[^A-Za-z0-9_]+")
```

Then each call to `next` returns an identifier.



- E15.6** Read all words from a file and add them to a map whose keys are the first letters of the words and whose values are sets of words that start with that same letter. Then print out the word sets in alphabetical order.

Provide two versions of your solution, one that uses the `merge` method (see Java 8 Note 15.1) and one that updates the map as in Worked Example 15.1.



- E15.7** Read all words from a file and add them to a map whose keys are word lengths and whose values are comma-separated strings of words of the same length. Then print out those strings, in increasing order by the length of their entries.

Provide two versions of your solution, one that uses the `merge` method (see Java 8 Note 15.1) and one that updates the map as in Worked Example 15.1.

- E15.8** Use a stack to reverse the words of a sentence. Keep reading words until you have a word that ends in a period, adding them onto a stack. When you have a word with a period, pop the words off and print them. Stop when there are no more words in the input. For example, you should turn the input

Mary had a little lamb. Its fleece was white as snow.

into

Lamb little a had mary. Snow as white was fleece its.

Pay attention to capitalization and the placement of the period.

- E15.9** Your task is to break a number into its individual digits, for example, to turn 1729 into 1, 7, 2, and 9. It is easy to get the last digit of a number  $n$  as  $n \% 10$ . But that gets the numbers in reverse order. Solve this problem with a stack. Your program should ask the user for an integer, then print its digits separated by spaces.

- E15.10** A homeowner rents out parking spaces in a driveway during special events. The driveway is a “last-in, first-out” stack. Of course, when a car owner retrieves a vehicle that wasn’t the last one in, the cars blocking it must temporarily move to the street so that the requested vehicle can leave. Write a program that models this behavior, using one stack for the driveway and one stack for the street. Use integers as license plate numbers. Positive numbers add a car, negative numbers remove a car, zero stops the simulation. Print out the stack after each operation is complete.

- E15.11** Implement a to do list. Tasks have a priority between 1 and 9, and a description. When the user enters the command `add priority description`, the program adds a new task. When the user enters `next`, the program removes and prints the most urgent task. The `quit` command quits the program. Use a priority queue in your solution.

- E15.12** Write a program that reads text from a file and breaks it up into individual words. Insert the words into a tree set. At the end of the input file, print all words, followed by the size of the resulting set. This program determines how many unique words a text file has.

- E15.13** Insert all words from a large file (such as the novel “War and Peace”, which is available on the Internet) into a hash set and a tree set. Time the results. Which data structure is more efficient?

- E15.14** Supply compatible `hashCode` and `equals` methods to the `BankAccount` class of Chapter 8. Test the `hashCode` method by printing out hash codes and by adding `BankAccount` objects to a hash set.

- E15.15** A labeled point has  $x$ - and  $y$ -coordinates and a string label. Provide a class `LabeledPoint` with a constructor `LabeledPoint(int x, int y, String label)` and `hashCode` and `equals` methods. Two labeled points are considered the same when they have the same location and label.
- E15.16** Reimplement the `LabeledPoint` class of Exercise E15.15 by storing the location in a `java.awt.Point` object. Your `hashCode` and `equals` methods should call the `hashCode` and `equals` methods of the `Point` class.
- E15.17** Modify the `LabeledPoint` class of Exercise E15.15 so that it implements the `Comparable` interface. Sort points first by their  $x$ -coordinates. If two points have the same  $x$ -coordinate, sort them by their  $y$ -coordinates. If two points have the same  $x$ - and  $y$ -coordinates, sort them by their label. Write a tester program that checks all cases by inserting points into a `TreeSet`.
- E15.18** Add a `%` (remainder) operator to the expression calculator of Section 15.6.3.
- E15.19** Add a `^` (power) operator to the expression calculator of Section 15.6.3. For example,  $2 ^ 3$  evaluates to 8. As in mathematics, your power operator should be evaluated from the right. That is,  $2 ^ 3 ^ 2$  is  $2 ^ (3 ^ 2)$ , not  $(2 ^ 3) ^ 2$ . (That's more useful because you could get the latter as  $2 ^ (3 \times 2)$ .)
- E15.20** Write a program that checks whether a sequence of HTML tags is properly nested. For each opening tag, such as `<p>`, there must be a closing tag `</p>`. A tag such as `<p>` may have other tags inside, for example
 

```
<p> <a> </p>
```

 The inner tags must be closed before the outer ones. Your program should process a file containing tags. For simplicity, assume that the tags are separated by spaces, and that there is no text inside the tags.
- E15.21** Modify the maze solver program of Section 15.6.4 to handle mazes with cycles. Keep a set of visited intersections. When you have previously seen an intersection, treat it as a dead end and do not add paths to the stack.
- E15.22** In a paint program, a “flood fill” fills all empty pixels of a drawing with a given color, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a  $10 \times 10$  array of integers that are initially 0.
 

**Prompt for the starting row and column.**

**Push the (row, column) pair onto a stack.**

You will need to provide a simple `Pair` class.

Repeat the following operations until the stack is empty.

**Pop off the (row, column) pair from the top of the stack.**

**If it has not yet been filled, fill the corresponding array location with a number 1, 2, 3, and so on (to show the order in which the square is filled).**

**Push the coordinates of any unfilled neighbors in the north, east, south, or west direction on the stack.**

When you are done, print the entire array.



## PROGRAMMING PROJECTS

- P15.1** Read all words from a list of words and add them to a map whose keys are the phone keypad spellings of the word, and whose values are sets of words with the same code. For example, 26337 is mapped to the set { "Andes", "coder", "codes", . . . }. Then keep prompting the user for numbers and print out all words in the dictionary that can be spelled with that number. In your solution, use a map that maps letters to digits.



© klenger/Stockphoto

- P15.2** Reimplement Exercise E15.4 so that the keys of the map are objects of class `Student`. A student should have a first name, a last name, and a unique integer ID. For grade changes and removals, lookup should be by ID. The printout should be sorted by last name. If two students have the same last name, then use the first name as a tie breaker. If the first names are also identical, then use the integer ID. *Hint:* Use two maps.

- P15.3** Write a class `Polynomial` that stores a polynomial such as

$$p(x) = 5x^{10} + 9x^7 - x - 10$$

as a linked list of terms. A term contains the coefficient and the power of  $x$ . For example, you would store  $p(x)$  as

$$(5,10), (9,7), (-1,1), (-10,0)$$

Supply methods to add, multiply, and print polynomials. Supply a constructor that makes a polynomial from a single term. For example, the polynomial `p` can be constructed as

```
Polynomial p = new Polynomial(new Term(-10, 0));
p.add(new Polynomial(new Term(-1, 1)));
p.add(new Polynomial(new Term(9, 7)));
p.add(new Polynomial(new Term(5, 10)));
```

Then compute  $p(x) \times p(x)$ .

```
Polynomial q = p.multiply(p);
q.print();
```

- P15.4** Repeat Exercise P15.3, but use a `Map<Integer, Double>` for the coefficients.

- P15.5** Try to find two words with the same hash code in a large file. Keep a `Map<Integer, HashSet<String>>`. When you read in a word, compute its hash code  $h$  and put the word in the set whose key is  $h$ . Then iterate through all keys and print the sets whose size is greater than one.

- P15.6** Supply compatible `hashCode` and `equals` methods to the `Student` class described in Exercise P15.2. Test the hash code by adding `Student` objects to a hash set.

- P15.7** Modify the expression calculator of Section 15.6.3 to convert an expression into reverse Polish notation. *Hint:* Instead of evaluating the top and pushing the result, append the instructions to a string.

- P15.8** Repeat Exercise E15.22, but use a queue instead.

- P15.9** Use a stack to enumerate all permutations of a string. Suppose you want to find all permutations of the string `meat`.

```

Push the string +meat on the stack.
While the stack is not empty
 Pop off the top of the stack.
 If that string ends in a + (such as tame+)
 Remove the + and add the string to the list of permutations.
 Else
 Remove each letter in turn from the right of the +.
 Insert it just before the +.
 Push the resulting string on the stack.

```

For example, after popping `e+mta`, you push `em+ta`, `et+ma`, and `ea+mt`.

- P15.10** Repeat Exercise P15.9, but use a queue instead.

- Business P15.11** An airport has only one runway. When it is busy, planes wishing to take off or land have to wait. Implement a simulation, using two queues, one each for the planes waiting to take off and land. Landing planes get priority. The user enters commands `takeoff flightSymbol`, `land flightSymbol`, `next`, and `quit`. The first two commands place the flight in the appropriate queue. The next command finishes the current takeoff or landing and enables the next one, printing the action (takeoff or land) and the flight symbol.

- Business P15.12** Suppose you buy 100 shares of a stock at \$12 per share, then another 100 at \$10 per share, and then sell 150 shares at \$15. You have to pay taxes on the gain, but exactly what is the gain? In the United States, the FIFO rule holds: You first sell all shares of the first batch for a profit of \$300, then 50 of the shares from the second batch, for a profit of \$250, yielding a total profit of \$550. Write a program that can make these calculations for arbitrary purchases and sales of shares in a single company. The user enters commands `buy quantity price`, `sell quantity price` (which causes the gain to be displayed), and `quit`. Hint: Keep a queue of objects of a class `Block` that contains the quantity and price of a block of shares.

- Business P15.13** Extend Exercise P15.12 to a program that can handle shares of multiple companies. The user enters commands `buy symbol quantity price` and `sell symbol quantity price`. Hint: Keep a `Map<String, Queue<Block>>` that manages a separate queue for each stock symbol.

- Business P15.14** Consider the problem of finding the least expensive routes to all cities in a network from a given starting point. For example, in the network shown on the map on page 733, the least expensive route from Pendleton to Peoria has cost 8 (going through Pierre and Pueblo).

The following helper class expresses the distance to another city:

```

public class DistanceTo implements Comparable<DistanceTo>
{
 private String target;
 private int distance;

 public DistanceTo(String city, int dist) { target = city; distance = dist; }
 public String getTarget() { return target; }
 public int getDistance() { return distance; }
 public int compareTo(DistanceTo other) { return distance - other.distance; }
}

```

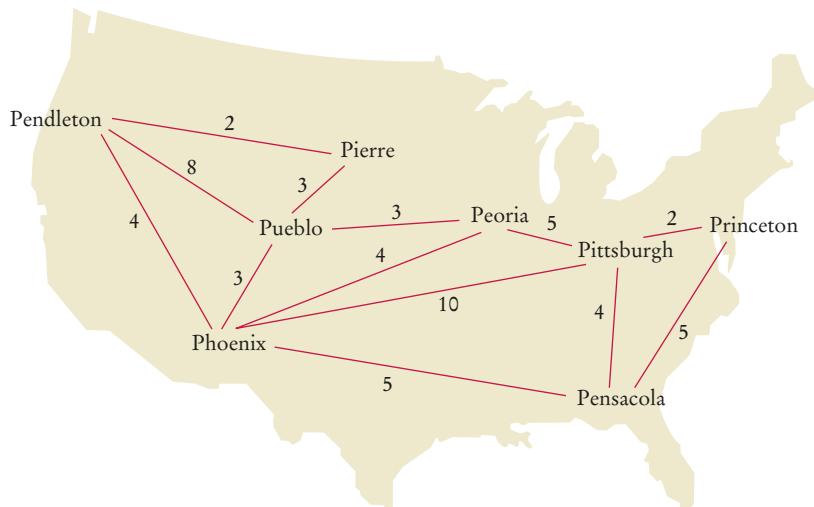
All direct connections between cities are stored in a `Map<String, HashSet<DistanceTo>>`. The algorithm now proceeds as follows:

```

Let from be the starting point.
Add DistanceTo(from, 0) to a priority queue.
Construct a map shortestKnownDistance from city names to distances.
While the priority queue is not empty
 Get its smallest element.
 If its target is not a key in shortestKnownDistance
 Let d be the distance to that target.
 Put (target, d) into shortestKnownDistance.
 For all cities c that have a direct connection from target
 Add DistanceTo(c, d + distance from target to c) to the priority queue.

```

When the algorithm has finished, `shortestKnownDistance` contains the shortest distance from the starting point to all reachable targets.



Your task is to write a program that implements this algorithm. Your program should read in lines of the form `city1 city2 distance`. The starting point is the first city in the first line. Print the shortest distances to all other cities.

### ANSWERS TO SELF-CHECK QUESTIONS

1. A list is a better choice because the application will want to retain the order in which the quizzes were given.
2. A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be efficient.
3. With a stack, you would always read the latest required reading, and you might never get to the oldest readings.
4. A collection stores elements, but a map stores associations between elements.
5. Yes, for two reasons. A linked list needs to store the neighboring node references, which are not needed in an array. Moreover, there is some overhead for storing an object. In a

linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

- linked list, each node is a separate object that incurs this overhead, whereas an array is a single object.

6. We can simply access each array element with an integer index.

7. ABCD  
A|BCD  
AB|CD  
A|CD  
AC|D  
ACE|D  
ACED|  
ACEDF|

8. 

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
 String str = iter.next();
 if (str.length() < 4) { iter.remove(); }
}
```

9. 

```
ListIterator<String> iter = words.iterator();
while (iter.hasNext())
{
 System.out.println(iter.next());
 if (iter.hasNext())
 {
 iter.next(); // Skip the next element
 }
}
```

10. Adding and removing elements as well as testing for membership is more efficient with sets.

11. Sets do not have an ordering, so it doesn't make sense to add an element at a particular iterator position, or to traverse a set backward.

12. You do not know in which order the set keeps the elements.

13. Here is one possibility:

```
if (s.size() == 3 && s.contains("Tom")
 && s.contains("Diana")
 && s.contains("Harry"))
 ...
14. for (String str : s)
{
 if (t.contains(str))
 {
 System.out.println(str);
 }
}
15. The words would be listed in sorted order.
16. A set stores elements. A map stores associations between keys and values.
```

17. The ordering does not matter, and you cannot have duplicates.

18. Because it might have duplicates.

19. Map<String, Integer> wordFrequency;  
Note that you cannot use a Map<String, int> because you cannot use primitive types as type parameters in Java.

20. It associates strings with sets of strings. One application would be a thesaurus that lists synonyms for a given word. For example, the key "improve" might have as its value the set ["ameliorate", "better", "enhance", "enrich", "perfect", "refine"].

21. This way, we can ensure that only queue operations can be invoked on the q object.

22. Depending on whether you consider the 0 position the head or the tail of the queue, you would either add or remove elements at that position. Both are inefficient operations because all other elements need to be moved.

23. A B C

24. Stacks use a "last-in, first-out" discipline. If you are the first one to submit a print job and lots of people add print jobs before the printer has a chance to deal with your job, they get their printouts first, and you have to wait until all other jobs are completed.

25. Yes—the smallest string (in lexicographic ordering) is removed first. In the example, that is the string starting with 1, then the string starting with 2, and so on. However, the scheme breaks down if a priority value exceeds 9; a string "10 - Line up braces" comes before "2 - Order supplies" in lexicographic order.

26. 70.

27. It would then subtract the first argument from the second. Consider the input 5 3 -. The stack contains 5 and 3, with the 3 on the top. Then results.pop() - results.pop() computes 3 - 5.

28. The - gets executed first because + doesn't have a higher precedence.

29. No, because there may be parentheses on the stack. The parentheses separate groups of operators, each of which is in increasing precedence.

30. A B E F G D C K J N

## WORKED EXAMPLE 15.1

**Word Frequency**

**Problem Statement** Write a program that reads a text file and prints a list of all words in the file in alphabetical order, together with a count that indicates how often each word occurred in the file.

For example, the following is the beginning of the output that results from processing the book *Alice in Wonderland*:

a	653
abide	1
able	1
about	97
above	4
absence	1
absurd	2

**Step 1** Determine how you access the values.

In our case, the values are the word frequencies. We have a frequency value for every word. That is, we want to use a map that maps words to frequencies.

**Step 2** Determine the element types of keys and values.

Each word is a String and each frequency is an Integer. (You cannot use an int as a type parameter because it is a primitive type.) Therefore, we need a Map<String, Integer>.

**Step 3** Determine whether element or key order matters.

We are supposed to print the words in sorted order, so we will use a TreeMap.

**Step 4** For a collection, determine which operations must be efficient.

We skip this step because we use a map, not a collection.

**Step 5** For hash sets and maps, decide what to do about the equals and hashCode methods.

We skip this step because we use a tree map.

**Step 6** If you use a tree, decide whether to supply a comparator.

The key type for our tree map is String, which implements the Comparable interface. Therefore, we need to do nothing further.

We have now chosen our collection. The program for completing our task is fairly simple. Here is the pseudocode:

```

For each word in the input file
 Remove non-letters (such as punctuation marks) from the word.
 If the word is already present in the frequencies map
 Increment the frequency.
 Else
 Set the frequency to 1.

```

Here is the program code:

### worked\_example\_1/WordFrequency.java

```

1 import java.util.Map;
2 import java.util.Scanner;
3 import java.util.TreeMap;
4 import java.io.File;
5 import java.io.FileNotFoundException;

```

```
6 /**
7 * This program prints the frequencies of all words in “Alice in Wonderland”.
8 */
9 public class WordFrequency
10 {
11 public static void main(String[] args)
12 throws FileNotFoundException
13 {
14 Map<String, Integer> frequencies = new TreeMap<>();
15 Scanner in = new Scanner(new File("alice30.txt"));
16 while (in.hasNext())
17 {
18 String word = clean(in.next());
19
20 // Get the old frequency count
21
22 Integer count = frequencies.get(word);
23
24 // If there was none, put 1; otherwise, increment the count
25
26 if (count == null) { count = 1; }
27 else { count = count + 1; }
28
29 frequencies.put(word, count);
30 }
31
32 // Print all words and counts
33
34 for (String key : frequencies.keySet())
35 {
36 System.out.printf("%-20s%10d%n", key, frequencies.get(key));
37 }
38 }
39
40 /**
41 * Removes characters from a string that are not letters.
42 * @param s a string
43 * @return a string with all the letters from s
44 */
45 public static String clean(String s)
46 {
47 String r = "";
48 for (int i = 0; i < s.length(); i++)
49 {
50 char c = s.charAt(i);
51 if (Character.isLetter(c))
52 {
53 r = r + c;
54 }
55 }
56 return r.toLowerCase();
57 }
58 }
59 }
```

## WORKED EXAMPLE 15.2

**Simulating a Queue of Waiting Customers**

A good application of object-oriented programming is simulation. In fact, the first object-oriented language, Simula, was designed with this application in mind. One can simulate the activities of air molecules around an aircraft wing, of customers in a supermarket, or of vehicles on a road system. The goal of a simulation is to observe how changes in the design affect the behavior of a system. Modifying the shape of a wing, the location and staffing of cash registers, or the synchronization of traffic lights has an effect on turbulence in the air stream, customer satisfaction, or traffic throughput. Modeling these systems in the computer is far cheaper than running actual experiments.

**Kinds of Simulation**

Simulations fall into two broad categories. A *continuous simulation* constantly updates all objects in a system. A simulated clock advances in seconds or some other suitable constant time interval. At every clock tick, each object is moved or updated in some way. Consider the simulation of traffic along a road. Each car has some position, velocity, and acceleration. Its position needs to be updated with every clock tick. If the car gets too close to an obstacle, it must decelerate. The new position may be displayed on the screen.

In contrast, in a *discrete event simulation*, time advances in chunks. All interesting events are kept in a priority queue, sorted by the time in which they are to happen. As soon as one event has completed, the clock jumps to the time of the next event to be executed.

To see the contrast between these two simulation styles, consider the updating of a traffic light. Suppose the traffic light just turned red, and it will turn green again in 30 seconds. In a continuous model, the traffic light is visited every second, and a counter variable is decremented. Once the counter reaches 0, the color changes. In a discrete model, the traffic light schedules an event to be notified 30 seconds from now. For 29 seconds, the traffic light is not bothered at all, and then it receives a message to change its state. Discrete event simulation avoids “busy waiting”.

In this Worked Example, you will see how to use queues and priority queues in a discrete event simulation of customers at a bank. The simulation makes use of two generic classes, `Event` and `Simulation`, that are useful for any discrete event simulation. We use inheritance to extend these classes to make classes that simulate the bank.

**Events**

A discrete event simulation generates, stores, and processes events. Each event has a time stamp indicating when it is to be executed. Each event has some action associated with it that must be carried out at that time. Beyond these properties, the scheduler has no concept of what an event represents. Of course, actual events must carry with them some information. For example, the event notifying a traffic light of a state change must know which traffic light to notify.

To do so, we will have all events extend a common superclass, `Event`. An `Event` object has an instance variable to indicate at which time it should be processed. When that time has arrived, the event’s `process` method is called. This method may move objects around, update information, and schedule additional events.

The `Event` class also implements the `Comparable` interface. An event is considered more urgent than another if its processing time is earlier.

```
public class Event implements Comparable<Event>
{
 private double time;

 public Event(double eventTime)
 {
```

```

 time = eventTime;
 }

 public void process(Simulation sim) {}
 public double getTime() { return time; }

 public int compareTo(Event other)
 {
 if (time < other.time) { return -1; }
 else if (time > other.time) { return 1; }
 else { return 0; }
 }
}

```

### The Simulation Class

In any discrete event simulation, events are kept in a priority queue. After initialization, the simulation enters an event loop in which events are retrieved from the priority queue in the order specified by their time stamp. The simulated time is advanced to the time stamp of the event, and the event is processed according to its process method. To simulate a specific activity, such as customer activity in a bank, extend the `Simulation` class and provide methods for displaying the current state after each event, and a summary after the completion of the simulation.

```

public class Simulation
{
 private PriorityQueue<Event> eventQueue;
 private double currentTime;
 . .
 public void display() {}
 public void displaySummary() {}
 . .
}

```

Here is the event loop in the `Simulation` class:

```

public void run(double startTime, double endTime)
{
 currentTime = startTime;

 while (eventQueue.size() > 0 && currentTime <= endTime)
 {
 Event event = eventQueue.remove();
 currentTime = event.getTime();
 event.process(this);
 display();
 }
 displaySummary();
}

```

In the `Simulation` class, we provide a utility method for generating reasonable random values for the time between two independent events. These random time differences can be modeled with an “exponential distribution”, as follows: Let  $m$  be the mean time between arrivals. Let  $u$  be a random value that can, with equal probability, assume any floating-point value between 0 inclusive and 1 exclusive. Then inter-arrival times can be generated as

$$a = -m \log(1 - u)$$

where  $\log$  is the natural logarithm. The utility method `expdist` computes these random values:

```

public static double expdist(double mean)
{
 return -mean * Math.log(1 - Math.random());
}

```

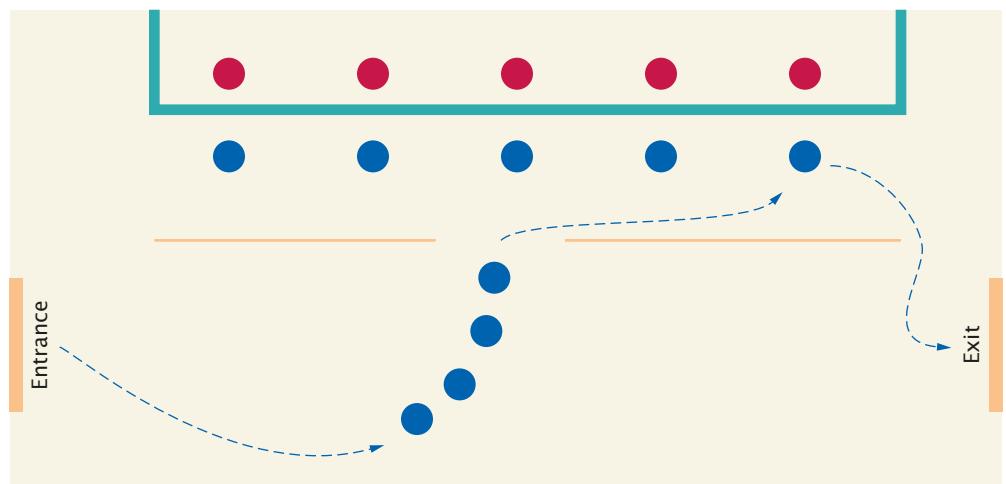
```
}
```

If a customer arrives at time  $t$ , the program can schedule the next customer arrival at  $t + \text{expdist}(m)$ .

Processing time is also exponentially distributed, with a different average. In this simulation we assume that, on average, one minute elapses between customer arrivals, and customer transactions require an average of five minutes.

### The Bank

The following figure shows the layout of the bank. Customers enter the bank. If there is a queue, they join the queue; otherwise they move up to a teller. When a customer has completed a teller transaction, the time spent in the bank is logged, the customer is removed, and the next customer in the queue moves up to the teller.



The `BankSimulation` class keeps an array of tellers as well as a queue to hold waiting customers. The queue is not a priority queue but a regular FIFO (first-in, first-out) queue:

```
public class BankSimulation extends Simulation
{
 private Customer[] tellers;
 private Queue<Customer> custQueue;

 private int totalCustomers;
 private double totalTime;

 private static final double INTERARRIVAL = 1;
 // average of 1 minute between customer arrivals
 private static final double PROCESSING = 5;
 // average of 5 minutes processing time per customer
 . .
}
```

It also keeps track of the total number of customers that have been serviced, and the total amount of time they spent in the bank (both in the waiting queue and in front of a teller.)

Teller  $i$  is busy if `tellers[i]` holds a reference to a `Customer` object and available if it is `null`.

When a customer is added to the bank, the program first checks whether a teller is available to handle the customer. If not, the customer is added to the waiting queue:

```

public void add(Customer c)
{
 boolean addedToTeller = false;
 for (int i = 0; !addedToTeller && i < tellers.length; i++)
 {
 if (tellers[i] == null)
 {
 addToTeller(i, c);
 addedToTeller = true;
 }
 }
 if (!addedToTeller) { custQueue.add(c); }

 addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
}

```

In addition, the simulation must ensure that customers keep coming. We know the next customer will arrive in about one minute, but it may be a bit earlier or, occasionally, a lot later. To obtain a random time, we call `expdist(INTERARRIVAL)`. Of course, we cannot wait around for that to happen, because other events will be going on in the meantime. Therefore when a customer is added, another arrival event is scheduled to occur when this random time has elapsed.

Similarly, when a customer steps up to a teller, the average transaction will be five minutes. We need to schedule a departure event that removes the customer from the bank. This happens in the `addToTeller` method:

```

private void addToTeller(int i, Customer c)
{
 tellers[i] = c;
 addEvent(new Departure(getCurrentTime() + expdist(PRECESSING), i));
}

```

When the departure event is processed, it will notify the bank to remove the customer. The bank simulation removes the customer and keeps track of the total amount of time the customer spent in the waiting queue and with the teller. This makes the teller available to service the next customer from the waiting queue. If there is a queue, we add the first customer to this teller:

```

public void remove(int i)
{
 Customer c = tellers[i];
 tellers[i] = null;

 // Update statistics
 totalCustomers++;
 totalTime = totalTime + getCurrentTime() - c.getArrivalTime();

 if (custQueue.size() > 0)
 {
 addToTeller(i, custQueue.remove());
 }
}

```

## Event Classes

The classes `Arrival` and `Departure` are subclasses of `Event`.

When a new customer is to arrive at the bank, an arrival event is processed. The processing action of that event has the responsibility of making a customer and adding it to the bank.

```

public class Arrival extends Event
{

```

```

public Arrival(double time) { super(time); }

public void process(Simulation sim)
{
 double now = sim.getCurrentTime();
 BankSimulation bank = (BankSimulation) sim;
 Customer c = new Customer(now);
 bank.add(c);
}
}

```

Departures remember not only the departure time but also the teller from whom a customer is to depart. To process a departure event, we remove the customer from the teller.

```

public class Departure extends Event
{
 private int teller;

 public Departure(double time, int teller)
 {
 super(time);
 this.teller = teller;
 }
 public void process(Simulation sim)
 {
 BankSimulation bank = (BankSimulation) sim;
 bank.remove(teller);
 }
}

```

## Running the Simulation

To run the simulation, we first construct a `BankSimulation` object with five tellers. The most important task in setting up the simulation is to get the flow of events going. At the outset, the event queue is empty. We will schedule the arrival of a customer at the start time (9 A.M.). Because the processing of an arrival event schedules the arrival of each successor, the insertion of the arrival event for the first customer takes care of the generation of all arrivals. Once customers arrive at the bank, they are added to tellers, and departure events are generated.

Here is the `main` method:

```

public static void main(String[] args)
{
 final double START_TIME = 9 * 60; // 9 A.M.
 final double END_TIME = 17 * 60; // 5 P.M.

 final int NTELLERS = 5;

 Simulation sim = new BankSimulation(NTELLERS);
 sim.addEvent(new Arrival(START_TIME));
 sim.run(START_TIME, END_TIME);
}

```

Here is a typical program run. The bank starts out with empty tellers, and customers start dropping in:

```

.....<
C....<
CC...<
CCC..<
CCCC.<
C.CC.<

```

```
CCCC.<
CCCCC<
CCCCC<C
CCCCC<
C.CCC<
```

Due to the random fluctuations of customer arrival and processing, the queue can get quite long:

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
```

```
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
CCCCC<CCCCCCCCCC
```

At other times, the bank is empty again:

```
CCC.C<
CCC..<
CC...<
.C...<
....<
C....<
```

This particular run of the simulation ends up with the following statistics:

```
457 customers. Average time 15.28 minutes.
```

If you are the bank manager, this result is quite depressing. You hired enough tellers to take care of all customers. (Every hour, you need to serve, on average, 60 customers. Their transactions take an average of 5 minutes each; that is 300 teller-minutes, or 5 teller-hours. Hence, hiring five tellers should be just right.) Yet the average customer had to wait in line more than 10 minutes, twice as long as their transaction time. This is an average, so some customers had to wait even longer. If disgruntled customers hurt your business, you may have to hire more tellers and pay them for being idle some of the time.

(See the ch15/worked\_example\_2 folder in your companion code for the complete bank simulation program.)

### **worked\_example\_2/BankSimulation.java**

```
1 import java.util.LinkedList;
2 import java.util.Queue;
3
4 /**
5 * Simulation of customer traffic in a bank.
6 */
7 public class BankSimulation extends Simulation
8 {
9 private Customer[] tellers;
10 private Queue<Customer> custQueue;
11
12 private int totalCustomers;
13 private double totalTime;
14
15 private static final double INTERARRIVAL = 1;
16 // average of 1 minute between customer arrivals
```

```

17 private static final double PROCESSING = 5;
18 // average of 5 minutes processing time per customer
19
20 public BankSimulation(int numberOfTellers)
21 {
22 tellers = new Customer[numberOfTellers];
23 custQueue = new LinkedList<>();
24 totalCustomers = 0;
25 totalTime = 0;
26 }
27
28 /**
29 * Adds a customer to the bank.
30 * @param c the customer
31 */
32 public void add(Customer c)
33 {
34 boolean addedToTeller = false;
35 for (int i = 0; !addedToTeller && i < tellers.length; i++)
36 {
37 if (tellers[i] == null)
38 {
39 addToTeller(i, c);
40 addedToTeller = true;
41 }
42 }
43 if (!addedToTeller) { custQueue.add(c); }
44
45 addEvent(new Arrival(getCurrentTime() + expdist(INTERARRIVAL)));
46 }
47
48 /**
49 * Adds a customer to a teller and schedules the departure event.
50 * @param i the teller number
51 * @param c the customer
52 */
53 private void addToTeller(int i, Customer c)
54 {
55 tellers[i] = c;
56 addEvent(new Departure(getCurrentTime() + expdist(PROCESSING), i));
57 }
58
59 /**
60 * Removes a customer from a teller.
61 * @param i teller position
62 */
63 public void remove(int i)
64 {
65 Customer c = tellers[i];
66 tellers[i] = null;
67
68 // Update statistics
69 totalCustomers++;
70 totalTime = totalTime + getCurrentTime() - c.getArrivalTime();
71
72 if (custQueue.size() > 0)
73 {
74 addToTeller(i, custQueue.remove());
75 }
76 }

```

```
77 /**
78 * Displays tellers and queue.
79 */
80 public void display()
81 {
82 for (int i = 0; i < tellers.length; i++)
83 {
84 if (tellers[i] == null)
85 {
86 System.out.print(".");
87 }
88 else
89 {
90 System.out.print("C");
91 }
92 }
93 System.out.print("<");
94 int q = custQueue.size();
95 for (int j = 1; j <= q; j++) { System.out.print("C"); }
96 System.out.println();
97 }
98
99
100 /**
101 * Displays a summary of the gathered statistics.
102 */
103 public void displaySummary()
104 {
105 double averageTime = 0;
106 if (totalCustomers > 0)
107 {
108 averageTime = totalTime / totalCustomers;
109 }
110 System.out.println(totalCustomers + " customers. Average time "
111 + averageTime + " minutes.");
112 }
113 }
```

# THE BASIC LATIN AND LATIN-1 SUBSETS OF UNICODE

This appendix lists the Unicode characters that are most commonly used for processing Western European languages. A complete listing of Unicode characters can be found at <http://unicode.org>.

Table 1 Selected Control Characters

Character	Code	Decimal	Escape Sequence
Tab	'\u0009'	9	'\t'
Newline	'\u000A'	10	'\n'
Return	'\u000D'	13	'\r'
Space	'\u0020'	32	

## A-2 Appendix A The Basic Latin and Latin-1 Subsets of Unicode

**Table 2 The Basic Latin (ASCII) Subset of Unicode**

Char.	Code	Dec.	Char.	Code	Dec.	Char.	Code	Dec.
			@	'\u0040'	64	`	'\u0060'	96
!	'\u0021'	33	A	'\u0041'	65	a	'\u0061'	97
"	'\u0022'	34	B	'\u0042'	66	b	'\u0062'	98
#	'\u0023'	35	C	'\u0043'	67	c	'\u0063'	99
\$	'\u0024'	36	D	'\u0044'	68	d	'\u0064'	100
%	'\u0025'	37	E	'\u0045'	69	e	'\u0065'	101
&	'\u0026'	38	F	'\u0046'	70	f	'\u0066'	102
'	'\u0027'	39	G	'\u0047'	71	g	'\u0067'	103
(	'\u0028'	40	H	'\u0048'	72	h	'\u0068'	104
)	'\u0029'	41	I	'\u0049'	73	i	'\u0069'	105
*	'\u002A'	42	J	'\u004A'	74	j	'\u006A'	106
+	'\u002B'	43	K	'\u004B'	75	k	'\u006B'	107
,	'\u002C'	44	L	'\u004C'	76	l	'\u006C'	108
-	'\u002D'	45	M	'\u004D'	77	m	'\u006D'	109
.	'\u002E'	46	N	'\u004E'	78	n	'\u006E'	110
/	'\u002F'	47	O	'\u004F'	79	o	'\u006F'	111
0	'\u0030'	48	P	'\u0050'	80	p	'\u0070'	112
1	'\u0031'	49	Q	'\u0051'	81	q	'\u0071'	113
2	'\u0032'	50	R	'\u0052'	82	r	'\u0072'	114
3	'\u0033'	51	S	'\u0053'	83	s	'\u0073'	115
4	'\u0034'	52	T	'\u0054'	84	t	'\u0074'	116
5	'\u0035'	53	U	'\u0055'	85	u	'\u0075'	117
6	'\u0036'	54	V	'\u0056'	86	v	'\u0076'	118
7	'\u0037'	55	W	'\u0057'	87	w	'\u0077'	119
8	'\u0038'	56	X	'\u0058'	88	x	'\u0078'	120
9	'\u0039'	57	Y	'\u0059'	89	y	'\u0079'	121
:	'\u003A'	58	Z	'\u005A'	90	z	'\u007A'	122
;	'\u003B'	59	[	'\u005B'	91	{	'\u007B'	123
<	'\u003C'	60	\	'\u005C'	92		'\u007C'	124
=	'\u003D'	61	]	'\u005D'	93	}	'\u007D'	125
>	'\u003E'	62	^	'\u005E'	94	~	'\u007E'	126
?	'\u003F'	63	_	'\u005F'	95			

**Table 3** The Latin-1 Subset of Unicode

Char.	Code	Dec.	Char.	Code	Dec.	Char.	Code	Dec.
			À	'\u00C0'	192	à	'\u00E0'	224
í	'\u00A1'	161	Á	'\u00C1'	193	á	'\u00E1'	225
¢	'\u00A2'	162	Â	'\u00C2'	194	â	'\u00E2'	226
£	'\u00A3'	163	Ã	'\u00C3'	195	ã	'\u00E3'	227
¤	'\u00A4'	164	Ä	'\u00C4'	196	ä	'\u00E4'	228
¥	'\u00A5'	165	Å	'\u00C5'	197	å	'\u00E5'	229
¡	'\u00A6'	166	Æ	'\u00C6'	198	æ	'\u00E6'	230
§	'\u00A7'	167	Ç	'\u00C7'	199	ç	'\u00E7'	231
„	'\u00A8'	168	È	'\u00C8'	200	è	'\u00E8'	232
©	'\u00A9'	169	É	'\u00C9'	201	é	'\u00E9'	233
ª	'\u00AA'	170	Ê	'\u00CA'	202	ê	'\u00EA'	234
«	'\u00AB'	171	Ë	'\u00CB'	203	ë	'\u00EB'	235
¬	'\u00AC'	172	Ì	'\u00CC'	204	ì	'\u00EC'	236
-	'\u00AD'	173	Í	'\u00CD'	205	í	'\u00ED'	237
®	'\u00AE'	174	Î	'\u00CE'	206	î	'\u00EE'	238
-	'\u00AF'	175	Ï	'\u00CF'	207	ï	'\u00EF'	239
º	'\u00B0'	176	Ð	'\u00D0'	208	ð	'\u00F0'	240
±	'\u00B1'	177	Ñ	'\u00D1'	209	ñ	'\u00F1'	241
²	'\u00B2'	178	Ò	'\u00D2'	210	ò	'\u00F2'	242
³	'\u00B3'	179	Ó	'\u00D3'	211	ó	'\u00F3'	243
›	'\u00B4'	180	Ô	'\u00D4'	212	ô	'\u00F4'	244
µ	'\u00B5'	181	Õ	'\u00D5'	213	õ	'\u00F5'	245
¶	'\u00B6'	182	Ö	'\u00D6'	214	ö	'\u00F6'	246
·	'\u00B7'	183	×	'\u00D7'	215	÷	'\u00F7'	247
,	'\u00B8'	184	Ø	'\u00D8'	216	ø	'\u00F8'	248
¹	'\u00B9'	185	Ù	'\u00D9'	217	ù	'\u00F9'	249
º	'\u00BA'	186	Ú	'\u00DA'	218	ú	'\u00FA'	250
»	'\u00BB'	187	Û	'\u00DB'	219	û	'\u00FB'	251
¼	'\u00BC'	188	Ü	'\u00DC'	220	ü	'\u00FC'	252
½	'\u00BD'	189	Ý	'\u00DD'	221	ý	'\u00FD'	253
¾	'\u00BE'	190	Þ	'\u00DE'	222	þ	'\u00FE'	254
¿	'\u00BF'	191	ß	'\u00DF'	223	ÿ	'\u00FF'	255



# JAVA OPERATOR SUMMARY

The Java operators are listed in groups of decreasing *precedence* in the table below. The horizontal lines in the table indicate a change in operator precedence. Operators with higher precedence bind more strongly than those with lower precedence. For example,  $x + y * z$  means  $x + (y * z)$  because the  $*$  operator has higher precedence than the  $+$  operator. Looking at the table below, you can tell that  $x \&& y || z$  means  $(x \&& y) || z$  because the  $||$  operator has lower precedence.

The *associativity* of an operator indicates whether it groups left to right, or right to left. For example, the  $-$  operator binds left to right. Therefore,  $x - y - z$  means  $(x - y) - z$ . But the  $=$  operator binds right to left, and  $x = y = z$  means  $x = (y = z)$ .

Operator	Description	Associativity
.	Access class feature	
[]	Array subscript	Left to right
()	Function call	
++	Increment	
--	Decrement	
!	Boolean <i>not</i>	
$\sim$	Bitwise <i>not</i>	
+ ( <i>unary</i> )	(Has no effect)	Right to left
- ( <i>unary</i> )	Negative	
( <i>TypeName</i> )	Cast	
new	Object allocation	
*	Multiplication	
/	Division or integer division	Left to right
%	Integer remainder	
+	Addition, string concatenation	
-	Subtraction	Left to right
<<	Shift left	
>>	Right shift with sign extension	Left to right
>>>	Right shift with zero extension	

## A-6 Appendix B Java Operator Summary

Operator	Description	Associativity
<	Less than	Left to right
<=	Less than or equal	
>	Greater than	
>=	Greater than or equal	
instanceof	Tests whether an object's type is a given type or a subtype thereof	
==	Equal	Left to right
!=	Not equal	
&	Bitwise <i>and</i>	Left to right
^	Bitwise exclusive <i>or</i>	Left to right
	Bitwise <i>or</i>	Left to right
&&	Boolean “short circuit” <i>and</i>	Left to right
	Boolean “short circuit” <i>or</i>	Left to right
? :	Conditional	Right to left
=	Assignment	Right to left
op=	Assignment with binary operator ( <i>op</i> is one of +, -, *, /, &,  , ^, <<, >>, >>>)	

# JAVA RESERVED WORD SUMMARY

Reserved Word	Description
abstract	An abstract class or method
assert	An assertion that a condition is fulfilled
boolean	The Boolean type
break	Breaks out of the current loop or labeled statement
byte	The 8-bit signed integer type
case	A label in a switch statement
catch	The handler for an exception in a try block
char	The 16-bit Unicode character type
class	Defines a class
const	Not used
continue	Skips the remainder of a loop body
default	The default label in a switch statement
do	A loop whose body is executed at least once
double	The 64-bit double-precision floating-point type
else	The alternative clause in an if statement
enum	An enumeration type
extends	Indicates that a class is a subclass of another class
final	A value that cannot be changed after it has been initialized, a method that cannot be overridden, or a class that cannot be extended
finally	A clause of a try block that is always executed
float	The 32-bit single-precision floating-point type
for	A loop with initialization, condition, and update expressions
goto	Not used
if	A conditional branch statement
implements	Indicates that a class realizes an interface

## A-8 Appendix C Java Reserved Word Summary

Reserved Word	Description
import	Allows the use of class names without the package name
instanceof	Tests whether an object's type is a given type or a subtype thereof
int	The 32-bit integer type
interface	An abstract type with only abstract or default methods and constants
long	The 64-bit integer type
native	A method implemented in non-Java code
new	Allocates an object
package	A collection of related classes
private	A feature that is accessible only by methods of the same class
protected	A feature that is accessible only by methods of the same class, a subclass, or another class in the same package
public	A feature that is accessible by all methods
return	Returns from a method
short	The 16-bit integer type
static	A feature that is defined for a class, not for individual instances
strictfp	Uses strict rules for floating-point computations
super	Invokes the superclass constructor or a superclass method
switch	A selection statement
synchronized	A block of code that is accessible to only one thread at a time
this	The implicit parameter of a method; or invocation of another constructor of the same class
throw	Throws an exception
throws	Indicates the exceptions that a method may throw
transient	Instance variables that should not be serialized
try	A block of code with exception handlers or a finally handler
void	Tags a method that doesn't return a value
volatile	A variable that may be accessed by multiple threads without synchronization
while	A loop statement

# THE JAVA LIBRARY

This appendix lists all classes and methods from the standard Java library that are used in this book. Classes are sorted first by package, then alphabetically within a package.

In the following inheritance hierarchy, superclasses that are not used in this book are shown in gray type. Some classes implement interfaces not covered in this book; they are omitted.

```
java.lang.AutoCloseable
java.lang.Object
 java.awt.BorderLayout
 java.awt.Color
 java.awt.Component
 java.awt.Container
 javax.swing.JComponent
 javax.swing.AbstractButton
 javax.swing.JButton
 javax.swing.JMenuItem
 javax.swing.JMenu
 javax.swing.JToggleButton
 javax.swing.JCheckBox
 javax.swing.JRadioButton
 javax.swing.JComboBox
 javax.swing.JFileChooser
 javax.swing.JLabel
 javax.swing.JMenuBar
 javax.swing.JPanel
 javax.swing.JOptionPane
 javax.swing.JScrollPane
 javax.swing.JSlider
 javax.swing.text.JTextComponent
 javax.swing.JTextArea
 javax.swing.JTextField
 java.awt.Window
 java.awt.Frame
 javax.swing.JFrame
 java.awt.Dimension2D
 java.awt.Dimension
 java.awt.FlowLayout
 java.awt.Font
 java.awt.Graphics
 java.awt.GridLayout
 java.awt.event.MouseAdapter implements MouseListener
 java.awt.geom.Point2D
 java.awt.geom.Point2D.Double
 java.awt.Point
 java.awt.geom.RectangularShape
 java.awt.geom.Rectangle2D
 java.awt.Rectangle
 java.io.File implements Comparable<File>
 java.io.InputStream
 java.io.FileInputStream
 java.io.OutputStream
 java.io.FileOutputStream
 java.io.FilterOutputStream
 java.io.PrintStream
```

## A-10 Appendix D The Java Library

```
java.io.Writer
 java.io.PrintWriter
java.lang.Boolean implements Comparable<Boolean>
java.lang.Character implements Comparable<Character>
java.lang.Class
java.lang.Math
java.lang.Number
 java.math.BigDecimal implements Comparable<BigDecimal>
 java.math.BigInteger implements Comparable<BigInteger>
 java.lang.Double implements Comparable<Double>
 java.lang.Integer implements Comparable<Integer>
java.lang.String implements Comparable<String>
java.lang.System
java.lang.Throwable
 java.lang.Error
 java.lang.Exception
 java.lang.InterruptedException
 java.io.IOException
 java.io.EOFException
 java.io.FileNotFoundException
 java.lang.RuntimeException
 java.lang.IllegalArgumentException
 java.lang.NumberFormatException
 java.lang.IllegalStateException
 java.util.NoSuchElementException
 java.util.InputMismatchException
 java.lang.NullPointerException
java.net.URL
java.util.AbstractCollection<E>
 java.util.AbstractList<E>
 java.util.AbstractSequentialList<E>
 java.util.LinkedList<E> implements List<E>, Queue<E>
 java.util.ArrayList<E> implements List<E>
 java.util.AbstractQueue<E>
 java.util.PriorityQueue<E>
 java.util.AbstractSet<E>
 java.util.HashSet<E> implements Set<E>
 java.util.TreeSet<E> implements SortedSet<E>
java.util.AbstractMap<K, V>
 java.util.HashMap<K, V> implements Map<K, V>
 java.util.LinkedHashMap<K, V>
 java.util.TreeMap<K, V> implements Map<K, V>
java.util.Arrays
java.util.Collections
java.util.Calendar
 java.util.GregorianCalendar
java.util.Date
java.util.EventObject
 java.awt.AWTEvent
 java.awt.event.ActionEvent
 java.awt.event.ComponentEvent
 java.awt.event.InputEvent
 java.awt.event.KeyEvent
 java.awt.event.MouseEvent
 javax.swing.event.ChangeEvent
java.util.Objects
java.util.Random
java.util.Scanner
java.util.logging.Level
java.util.logging.Logger
javax.swing.ButtonGroup
javax.swing.ImageIcon
javax.swing.KeyStroke
```

```

javax.swing.Timer
javax.swing.border.AbstractBorder
 javax.swing.border.EtchedBorder
 javax.swing.border.TitledBorder
java.lang.Comparable<T>
java.util.Collection<E>
 java.util.List<E>
 java.util.Set<E>
 java.util.SortedSet<E>
java.util.Comparator<T>
java.util.EventListener
 java.awt.event.ActionListener
 java.awt.event.KeyListener
 java.awt.eventMouseListener
 javax.swing.event.ChangeListener
java.util.Iterator<E>
 java.util.ListIterator<E>
java.util.Map<K, V>
java.util.Queue<E> extends Collection<E>

```

In the following descriptions, the phrase “this object” (“this component”, “this container”, and so forth) means the object (component, container, and so forth) on which the method is invoked (the implicit parameter, `this`).

## Package `java.awt`

### Class `java.awt.BorderLayout`

- **`BorderLayout()`**

This constructs a border layout. A border layout has five regions for adding components, called “NORTH”, “EAST”, “SOUTH”, “WEST”, and “CENTER”.

- **`static final int CENTER`**

This value identifies the center position of a border layout.

- **`static final int EAST`**

This value identifies the east position of a border layout.

- **`static final int NORTH`**

This value identifies the north position of a border layout.

- **`static final int SOUTH`**

This value identifies the south position of a border layout.

- **`static final int WEST`**

This value identifies the west position of a border layout.

### Class `java.awt.Color`

- **`Color(int red, int green, int blue)`**

This creates a color with the specified red, green, and blue values between 0 and 255.

**Parameters:** `red` The red component

`green` The green component

`blue` The blue component

### Class `java.awt.Component`

- **`void addKeyListener(KeyListener listener)`**

This method adds a key listener to the component.

**Parameters:** `listener` The key listener to be added

- **`void addMouseListener(MouseListener listener)`**

This method adds a mouse listener to the component.

**Parameters:** `listener` The mouse listener to be added

- **`int getHeight()`**

This method gets the height of this component.

**Returns:** The height in pixels

- **`int getWidth()`**

This method gets the width of this component.

**Returns:** The width in pixels

## A-12 Appendix D The Java Library

### • void repaint()

This method repaints this component by scheduling a call to the paint method.

### • void setFocusable(boolean focusable)

This method controls whether or not the component can receive input focus.

**Parameters:** focusable true to have focus, or false to lose focus

### • void setPreferredSize(Dimension preferredSize)

This method sets the preferred size of this component.

### • void setSize(int width, int height)

This method sets the size of this component.

**Parameters:** width the component width  
height the component height

### • void setVisible(boolean visible)

This method shows or hides the component.

**Parameters:** visible true to show the component, or false to hide it

## Class java.awt.Container

### • void add(Component c)

### • void add(Component c, Object position)

These methods add a component to the end of this container. If a position is given, the layout manager is called to position the component.

**Parameters:** c The component to be added  
position An object expressing position information for the layout manager

### • void setLayout(LayoutManager manager)

This method sets the layout manager for this container.

**Parameters:** manager A layout manager

## Class java.awt.Dimension

### • Dimension(int width, int height)

This constructs a Dimension object with the given width and height.

**Parameters:** width The width  
height The height

## Class java.awtFlowLayout

### • FlowLayout()

This constructs a new flow layout. A flow layout places as many components as possible in a row, without changing their size, and starts new rows as needed.

## Class java.awt.Font

### • Font(String name, int style, int size)

This constructs a font object from the specified name, style, and point size.

**Parameters:** name The font name, either a font face name or a logical font name, which must be one of "Dialog", "DialogInput", "Monospaced", "Serif", or "SansSerif"  
style One of Font.PLAIN, Font.ITALIC, Font.BOLD, or Font.ITALIC+Font.BOLD  
size The point size of the font

## Class java.awt.Frame

### • void setTitle(String title)

This method sets the frame title.

**Parameters:** title The title to be displayed in the border of the frame

## Class java.awt.Graphics

### • void drawLine(int x1, int y1, int x2, int y2)

Draws a line between two points.

**Parameters:** x1, y1 The starting point  
x2, y2 The endpoint

### • void drawOval(int x, int y, int width, int height)

### • void fillOval(int x, int y, int width, int height)

**Parameters:** x1, y1 The top-left corner of the bounding rectangle  
width, height The width and height of the bounding rectangle

### • void drawRect(int x, int y, int width, int height)

### • void fillRect(int x, int y, int width, int height)

**Parameters:** x1, y1 The top-left corner of the rectangle  
width, height The width and height of the rectangle

### • void drawString(String s, int x, int y)

This method draws a string in the current font and color.

**Parameters:** s The string to draw  
x, y The basepoint of the first character in the string

### • void setColor(Color c)

This method sets the current color. After the method call, all graphics operations use this color.

**Parameters:** c The new drawing color

**Class java.awt.GridLayout****• GridLayout(int rows, int cols)**

This constructor creates a grid layout with the specified number of rows and columns. The components in a grid layout are arranged in a grid with equal widths and heights. One, but not both, of rows and cols can be zero, in which case any number of objects can be placed in a row or in a column, respectively.

**Parameters:** rows The number of rows in the grid  
cols The number of columns in the grid

**Class java.awt.Point****• Point()**

This constructs a Point object at the origin (0, 0).

**• Point(int x, int y)**

This constructs a Point object at the specified (x, y) location in the coordinate space.

**Parameters:** x The x-coordinate  
y The y-coordinate

**• double getX()****• double getY()**

These methods get the x- and y-coordinates of the point as floating-point values.

**Class java.awt.Rectangle****• Rectangle()**

This constructs a rectangle with a top-left corner at (0, 0) and width and height set to 0.

**• Rectangle(int x, int y, int width, int height)**

This constructs a rectangle with given top-left corner and size.

**Parameters:** x, y The top-left corner  
width The width  
height The height

**• double getHeight()****• double getWidth()**

These methods get the height and width of the rectangle.

**• double getX()****• double getY()**

These methods get the x- and y-coordinates of the top-left corner of the rectangle.

**• void grow(int dw, int dh)**

This method adjusts the width and height of this rectangle.

**Parameters:** dw The amount to add to the width (can be negative)  
dh The amount to add to the height (can be negative)

**• Rectangle intersection(Rectangle other)**

This method computes the intersection of this rectangle with the specified rectangle.

**Parameters:** other A rectangle

**Returns:** The largest rectangle contained in both this and other

**• void setLocation(int x, int y)**

This method moves this rectangle to a new location.

**Parameters:** x, y The new top-left corner

**• void setSize(int width, int height)**

This method sets the width and height of this rectangle to new values.

**Parameters:** width The new width  
height The new height

**• void translate(int dx, int dy)**

This method moves this rectangle.

**Parameters:** dx The distance to move along the x-axis  
dy The distance to move along the y-axis

**• Rectangle union(Rectangle other)**

This method computes the union of this rectangle with the specified rectangle. This is not the set-theoretic union but the smallest rectangle that contains both this and other.

**Parameters:** other A rectangle

**Returns:** The smallest rectangle containing both this and other

**Package java.awt.event****Interface java.awt.event.ActionListener****• void actionPerformed(ActionEvent e)**

The event source calls this method when an action occurs.

**Class java.awt.event.KeyEvent**

This event is passed to the KeyListener methods. Use the KeyStroke class to obtain the key information from the key event.

## A-14 Appendix D The Java Library

### Interface `java.awt.event.KeyListener`

- `void keyPressed(KeyEvent e)`
- `void keyReleased(KeyEvent e)`

These methods are called when a key has been pressed or released.

- `void keyTyped(KeyEvent e)`

This method is called when a keystroke has been composed by pressing and releasing one or more keys.

### Class `java.awt.event.MouseEvent`

- `int getX()`

This method returns the horizontal position of the mouse as of the time the event occurred.

**Returns:** The *x*-position of the mouse

- `int getY()`

This method returns the vertical position of the mouse as of the time the event occurred.

**Returns:** The *y*-position of the mouse

### Interface `java.awt.event.MouseListener`

- `void mouseClicked(MouseEvent e)`

This method is called when the mouse has been clicked (that is, pressed and released in quick succession).

- `void mouseEntered(MouseEvent e)`

This method is called when the mouse has entered the component to which this listener was added.

- `void mouseExited(MouseEvent e)`

This method is called when the mouse has exited the component to which this listener was added.

- `void mousePressed(MouseEvent e)`

This method is called when a mouse button has been pressed.

- `void mouseReleased(MouseEvent e)`

This method is called when a mouse button has been released.

## Package `java.io`

### Class `java.io.EOFException`

- `EOFException(String message)`

This constructs an “end of file” exception object.

**Parameters:** `message` The detail message

### Class `java.io.File`

- `File(String name)`

This constructs a `File` object that describes a file (which may or may not exist) with the given name.  
**Parameters:** `name` The name of the file

- `boolean exists()`

This method checks whether there is a file in the local file system that matches this `File` object.

**Returns:** `true` if there is a matching file, `false` otherwise

- `static final String pathSeparator`

The system-dependent separator between path names. A colon (:) in Linux or Mac OS X; a semicolon (;) in Windows.

### Class `java.io.FileInputStream`

- `FileInputStream(File f)`

This constructs a file input stream and opens the chosen file. If the file cannot be opened for reading, a `FileNotFoundException` is thrown.

**Parameters:** `f` The file to be opened for reading

- `FileInputStream(String name)`

This constructs a file input stream and opens the named file. If the file cannot be opened for reading, a `FileNotFoundException` is thrown.

**Parameters:** `name` The name of the file to be opened for reading

### Class `java.io.FileNotFoundException`

This exception is thrown when a file could not be opened.

### Class `java.io.FileOutputStream`

- `FileOutputStream(File f)`

This constructs a file output stream and opens the chosen file. If the file cannot be opened for writing, a `FileNotFoundException` is thrown.

**Parameters:** `f` The file to be opened for writing

- `FileOutputStream(String name)`

This constructs a file output stream and opens the named file. If the file cannot be opened for writing, a `FileNotFoundException` is thrown.

**Parameters:** `name` The name of the file to be opened for writing

### Class `java.io.InputStream`

- `void close()`

This method closes this input stream (such as a `FileInputStream`) and releases any system resources associated with the stream.

- **int read()**

This method reads the next byte of data from this input stream.

**Returns:** The next byte of data, or -1 if the end of the stream is reached

### Class java.io.InputStreamReader

- **InputStreamReader(InputStream in)**

This constructs a reader from a specified input stream.

**Parameters:** in The stream to read from

### Class java.io.IOException

This type of exception is thrown when an input/output error is encountered.

### Class java.io.OutputStream

- **void close()**

This method closes this output stream (such as a `FileOutputStream`) and releases any system resources associated with this stream. A closed stream cannot perform output operations and cannot be reopened.

- **void write(int b)**

This method writes the lowest byte of b to this output stream.

**Parameters:** b The integer whose lowest byte is written

### Class java.io.PrintStream / Class java.io.PrintWriter

- **PrintStream(String name)**

- **PrintWriter(String name)**

This constructs a `PrintStream` or `PrintWriter` and opens the named file. If the file cannot be opened for writing, a `FileNotFoundException` is thrown.

**Parameters:** name The name of the file to be opened for writing

- **void close()**

This method closes this stream or writer and releases any associated system resources.

- **void print(int x)**

- **void print(double x)**

- **void print(Object x)**

- **void print(String x)**

- **void println()**

- **void println(int x)**

- **void println(double x)**

- **void println(Object x)**

- **void println(String x)**

These methods print a value to this `PrintStream` or `PrintWriter`. The `println` methods print a newline

after the value. Objects are printed by converting them to strings with their `toString` methods.

**Parameters:** x The value to be printed

- **PrintStream printf(String format, Object... values)**

- **PrintWriter printf(String format, Object... values)**

These methods print the format string to this `PrintStream` or `PrintWriter`, substituting the given values for placeholders that start with %.

**Parameters:** format The format string

values The values to be printed. You can supply any number of values

**Returns:** The implicit parameter

## Package java.lang

### Interface java.lang.AutoCloseable

- **void close()**

This method is called automatically at the end of a try-with-resources statement.

### Class java.lang.Boolean

- **Boolean(boolean value)**

This constructs a wrapper object for a boolean value.

**Parameters:** value The value to store in this object

- **boolean booleanValue()**

This method returns the value stored in this boolean object.

**Returns:** The Boolean value of this object

### Class java.lang.Character

- **static boolean isDigit(ch)**

This method tests whether a given character is a Unicode digit.

**Parameters:** ch The character to test

**Returns:** true if the character is a digit

- **static boolean isLetter(ch)**

This method tests whether a given character is a Unicode letter.

**Parameters:** ch The character to test

**Returns:** true if the character is a letter

- **static boolean isLowerCase(ch)**

This method tests whether a given character is a lowercase Unicode letter.

**Parameters:** ch The character to test

**Returns:** true if the character is a lowercase letter

## A-16 Appendix D The Java Library

- **static boolean isUpperCase(ch)**

This method tests whether a given character is an uppercase Unicode letter.

**Parameters:** ch The character to test

**Returns:** true if the character is an uppercase letter

### Class `java.lang.Class`

- **static Class forName(String className)**

This method loads a class with a given name. Loading a class initializes its static variables.

**Parameters:** className The name of the class to load

**Returns:** The type descriptor of the class

### Interface `java.lang.Comparable<T>`

- **int compareTo(T other)**

This method compares this object with the other object.

**Parameters:** other The object to be compared

**Returns:** A negative integer if this object is less than the other, zero if they are equal, or a positive integer otherwise

### Class `java.lang.Double`

- **Double(double value)**

This constructs a wrapper object for a double-precision floating-point number.

**Parameters:** value The value to store in this object

- **static int compare(double x, double y)**

This method compares two numbers.

**Parameters:** x, y Two floating-point values

**Returns:** A negative integer if x is less than y, zero if they are equal, or a positive integer otherwise

- **double doubleValue()**

This method returns the floating-point value stored in this Double wrapper object.

**Returns:** The value stored in the object

- **static double parseDouble(String s)**

This method returns the floating-point number that the string represents. If the string cannot be interpreted as a number, a NumberFormatException is thrown.

**Parameters:** s The string to be parsed

**Returns:** The value represented by the string argument

### Class `java.lang.Error`

This is the superclass for all unchecked system errors.

### Class `java.lang.IllegalArgumentException`

- **IllegalArgumentException()**

This constructs an IllegalArgumentException with no detail message.

### Class `java.lang.IllegalStateException`

This exception is thrown if the state of an object indicates that a method cannot currently be applied.

### Class `java.lang.Integer`

- **Integer(int value)**

This constructs a wrapper object for an integer.

**Parameters:** value The value to store in this object

- **static int compare(int x, int y)**

This method compares two numbers.

**Parameters:** x, y Two integer values

**Returns:** A negative integer if x is less than y, zero if they are equal, or a positive integer otherwise

- **int intValue()**

This method returns the integer value stored in this wrapper object.

**Returns:** The value stored in the object

- **static int parseInt(String s)**

This method returns the integer that the string represents. If the string cannot be interpreted as an integer, a NumberFormatException is thrown.

**Parameters:** s The string to be parsed

**Returns:** The value represented by the string argument

- **static Integer parseInt(String s, int base)**

This method returns the integer value that the string represents in a given number system. If the string cannot be interpreted as an integer, a NumberFormatException is thrown.

**Parameters:** s The string to be parsed

base The base of the number system (such as 2 or 16)

**Returns:** The value represented by the string argument

- **static String toString(int i)**

- **static String toString(int i, int base)**

This method creates a string representation of an integer in a given number system. If no base is given, a decimal representation is created.

**Parameters:** i An integer number

base The base of the number system (such as 2 or 16)

- Returns:** A string representation of the argument in the number system
- **static final int MAX\_VALUE**  
This constant is the largest value of type `int`.
  - **static final int MIN\_VALUE**  
This constant is the smallest (negative) value of type `int`.
- Class `java.lang.InterruptedException`**  
This exception is thrown to interrupt a thread, usually with the intention of terminating it.
- Class `java.lang.Math`**
- **static double abs(double x)**  
This method returns the absolute value  $|x|$ .  
**Parameters:** `x` A floating-point value  
**Returns:** The absolute value of the argument
  - **static double acos(double x)**  
This method returns the angle with the given cosine,  $\cos^{-1} x \in [0, \pi]$ .  
**Parameters:** `x` A floating-point value between  $-1$  and  $1$   
**Returns:** The arc cosine of the argument, in radians
  - **static double asin(double x)**  
This method returns the angle with the given sine,  $\sin^{-1} x \in [-\pi/2, \pi/2]$ .  
**Parameters:** `x` A floating-point value between  $-1$  and  $1$   
**Returns:** The arc sine of the argument, in radians
  - **static double atan(double x)**  
This method returns the angle with the given tangent,  $\tan^{-1} x \in (-\pi/2, \pi/2)$ .  
**Parameters:** `x` A floating-point value  
**Returns:** The arc tangent of the argument, in radians
  - **static double atan2(double y, double x)**  
This method returns the arc tangent,  $\tan^{-1}(y/x) \in (-\pi, \pi)$ . If `x` can equal zero, or if it is necessary to distinguish “northwest” from “southeast” and “northeast” from “southwest”, use this method instead of `atan(y/x)`.  
**Parameters:** `y, x` Two floating-point values  
**Returns:** The angle, in radians, between the points  $(0,0)$  and  $(x,y)$
  - **static double ceil(double x)**  
This method returns the smallest integer  $\geq x$  (as a `double`).  
**Parameters:** `x` A floating-point value  
**Returns:** The smallest integer greater than or equal to the argument

- **static double cos(double radians)**  
This method returns the cosine of an angle given in radians.  
**Parameters:** `radians` An angle, in radians  
**Returns:** The cosine of the argument
- **static double exp(double x)**  
This method returns the value  $e^x$ , where  $e$  is the base of the natural logarithms.  
**Parameters:** `x` A floating-point value  
**Returns:**  $e^x$
- **static double floor(double x)**  
This method returns the largest integer  $\leq x$  (as a `double`).  
**Parameters:** `x` A floating-point value  
**Returns:** The largest integer less than or equal to the argument
- **static int floorMod(int x, int y)**  
This method returns the “floor modulus” remainder of the integer division of `x` by `y`. If `y` is positive, the result is the (nonnegative) mathematical remainder.  
**Parameters:** `x, y` Two integers  
**Returns:** For positive `y`, the smallest nonnegative number `r` such that  $x = qy + r$  for some `r`. For negative `y`,  $-\text{Math.floorMod}(x, -y)$ .
- **static double log(double x)**
- **static double log10(double x)**  
This method returns the natural (base  $e$ ) or decimal (base 10) logarithm of `x`,  $\ln x$ .  
**Parameters:** `x` A number greater than 0.0  
**Returns:** The natural logarithm of the argument
- **static int max(int x, int y)**
- **static double max(double x, double y)**  
These methods return the larger of the given arguments.  
**Parameters:** `x, y` Two integers or floating-point values  
**Returns:** The maximum of the arguments
- **static int min(int x, int y)**
- **static double min(double x, double y)**  
These methods return the smaller of the given arguments.  
**Parameters:** `x, y` Two integers or floating-point values  
**Returns:** The minimum of the arguments
- **static double pow(double x, double y)**  
This method returns the value  $x^y$  ( $x > 0$ , or  $x = 0$  and  $y > 0$ , or  $x < 0$  and  $y$  is an integer).  
**Parameters:** `x, y` Two floating-point values  
**Returns:** The value of the first argument raised to the power of the second argument

## A-18 Appendix D The Java Library

- **static double random()**

This method returns a double value greater than or equal to 0.0 and less than 1.0.

**Returns:** A positive floating-point value  $\geq 0$  and  $< 1.0$

- **static long round(double x)**

This method returns the closest long integer to the argument.

**Parameters:**  $x$  A floating-point value

**Returns:** The argument rounded to the nearest long value

- **static double sin(double radians)**

This method returns the sine of an angle given in radians.

**Parameters:** radians An angle, in radians

**Returns:** The sine of the argument

- **static double sqrt(double x)**

This method returns the square root of  $x$ ,  $\sqrt{x}$ .

**Parameters:**  $x$  A nonnegative floating-point value

**Returns:** The square root of the argument

- **static double tan(double radians)**

This method returns the tangent of an angle given in radians.

**Parameters:** radians An angle, in radians

**Returns:** The tangent of the argument

- **static double toDegrees(double radians)**

This method converts radians to degrees.

**Parameters:** radians An angle, in radians

**Returns:** The angle in degrees

- **static double toRadians(double degrees)**

This method converts degrees to radians.

**Parameters:** degrees An angle, in degrees

**Returns:** The angle in radians

- **static final double E**

This constant is the value of  $e$ , the base of the natural logarithms.

- **static final double PI**

This constant is the value of  $\pi$ .

### Class `java.lang.NullPointerException`

This exception is thrown when a program tries to use an object through a null reference.

### Class `java.lang.NumberFormatException`

This exception is thrown when a program tries to parse the numerical value of a string that is not a number.

### Class `java.lang.Object`

- **protected Object clone()**

This constructs and returns a shallow copy of this object whose instance variables are copies of the instance variables of this object. If an instance variable of the object is an object reference itself, only the reference is copied, not the object itself. However, if the class does not implement the Cloneable interface, a CloneNotSupportedException is thrown. Subclasses should redefine this method to make a deep copy.

**Returns:** A copy of this object

- **boolean equals(Object other)**

This method tests whether this and the other object are equal. This method tests only whether the object references are to the same object. Subclasses should redefine this method to compare the instance variables.

**Parameters:** other The object with which to compare

**Returns:** true if the objects are equal, false otherwise

- **String toString()**

This method returns a string representation of this object. This method produces only the class name and locations of the objects. Subclasses should redefine this method to print the instance variables.

**Returns:** A string describing this object

### Class `java.lang.RuntimeException`

This is the superclass for all unchecked exceptions.

### Class `java.lang.String`

- **int compareTo(String other)**

This method compares this string and the other string lexicographically.

**Parameters:** other The other string to be compared

**Returns:** A value less than 0 if this string is lexicographically less than the other, 0 if the strings are equal, and a value greater than 0 otherwise

- **boolean equals(String other)**

- **boolean equalsIgnoreCase(String other)**

These methods test whether two strings are equal, or whether they are equal when letter case is ignored.

**Parameters:** other The other string to be compared

**Returns:** true if the strings are equal

- **static String format(String format, Object... values)**

This method formats the given string by substituting placeholders beginning with % with the given values.

**Parameters:** format The string with the placeholders  
values The values to be substituted for the placeholders

**Returns:** The formatted string, with the placeholders replaced by the given values

- **int length()**

This method returns the length of this string.

**Returns:** The count of characters in this string

- **String replace(String match, String replacement)**

This method replaces matching substrings with a given replacement.

**Parameters:** match The string whose matches are to be replaced  
replacement The string with which matching substrings are replaced

**Returns:** A string that is identical to this string, with all matching substrings replaced by the given replacement

- **String replaceAll(String regex, String replacement)**

This method replaces occurrences of a regular expression.

**Parameters:** regex A regular expression (see Special Topic 7.4)  
replacement The replacement string

**Returns:** A string in which all substrings matching regex are replaced with a replacement string

- **String[] split(String regex)**

This method splits a string around delimiters that match a regular expression.

**Parameters:** regex a regular expression (see Special Topic 7.4)

**Returns:** An array of strings that results from breaking this string along matches of regex. For example, "a,b;c".  
split("[,;]") yields an array of strings "a", "b", and "c".

- **String substring(int begin)**

- **String substring(int begin, int pastEnd)**

These methods return a new string that is a substring of this string, made up of all characters starting at position begin and up to either position pastEnd - 1, if it is given, or the end of the string.

**Parameters:** begin The beginning index, inclusive  
pastEnd The ending index, exclusive

**Returns:** The specified substring

- **String toLowerCase()**

This method returns a new string that consists of all characters in this string converted to lowercase.

**Returns:** A string with all characters in this string converted to lowercase

- **String toUpperCase()**

This method returns a new string that consists of all characters in this string converted to uppercase.

**Returns:** A string with all characters in this string converted to uppercase

## Class java.lang.System

- **static long currentTimeMillis()**

This method returns the difference, measured in milliseconds, between the current time and midnight, Universal Time, January 1, 1970.

**Returns:** The current time in milliseconds since January 1, 1970.

- **static void exit(int status)**

This method terminates the program.

**Parameters:** status Exit status. A nonzero status code indicates abnormal termination

- **static final InputStream in**

This object is the “standard input” stream. Reading from this stream typically reads keyboard input.

- **static final PrintStream out**

This object is the “standard output” stream.

Printing to this stream typically sends output to the console window.

## Class java.lang.Throwable

This is the superclass of exceptions and errors.

- **Throwable()**

This constructs a Throwable with no detail message.

- **String getMessage()**

This method gets the message that describes the exception or error.

**Returns:** The message

- **void printStackTrace()**

This method prints a stack trace to the “standard error” stream. The stack trace lists this object and all calls that were pending when it was created.

## Package `java.math`

### Class `java.math.BigDecimal`

- `BigDecimal(String value)`

This constructs an arbitrary-precision floating-point number from the digits in the given string.

**Parameters:** `value` A string representing the floating-point number

- `BigDecimal add(BigDecimal other)`
- `BigDecimal divide(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`

These methods return a `BigDecimal` whose value is the sum, difference, product, or quotient of this number and the other.

**Parameters:** `other` The other number

**Returns:** The result of the arithmetic operation

### Class `java.math.BigInteger`

- `BigInteger(String value)`

This constructs an arbitrary-precision integer from the digits in the given string.

**Parameters:** `value` A string representing an arbitrary-precision integer

- `BigInteger add(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger subtract(BigInteger other)`

These methods return a `BigInteger` whose value is the sum, quotient, remainder, product, or difference of this number and the other.

**Parameters:** `other` The other number

**Returns:** The result of the arithmetic operation

## Package `java.net`

### Class `java.net.URL`

- `URL(String s)`

This constructs a `URL` object from a string containing the URL.

**Parameters:** `s` The URL string, such as "http://horstmann.com/index.html"

- `InputStream openStream()`

This method gets the input stream through which the client can read the information that the server sends.

**Returns:** The input stream associated with this URL

## Package `java.util`

### Class `java.util.ArrayList<E>`

- `ArrayList()`

This constructs an empty array list.

- `boolean add(E element)`

This method appends an element to the end of this array list.

**Parameters:** `element` The element to add

**Returns:** `true` (This method returns a value because it overrides a method in the `List` interface.)

- `void add(int index, E element)`

This method inserts an element into this array list at the given position.

**Parameters:** `index` Insert position  
`element` The element to insert

- `E get(int index)`

This method gets the element at the specified position in this array list.

**Parameters:** `index` Position of the element to return

**Returns:** The requested element

- `E remove(int index)`

This method removes the element at the specified position in this array list and returns it.

**Parameters:** `index` Position of the element to remove

**Returns:** The removed element

- `E set(int index, E element)`

This method replaces the element at a specified position in this array list.

**Parameters:** `index` Position of element to replace  
`element` Element to be stored at the specified position

**Returns:** The element previously at the specified position

- `int size()`

This method returns the number of elements in this array list.

**Returns:** The number of elements in this array list

### Class `java.util.Arrays`

- `static int binarySearch(Object[] a, Object key)`

This method searches the specified array for the specified object using the binary search algorithm. The array elements must implement the `Comparable`

interface. The array must be sorted in ascending order.

**Parameters:** `a` The array to be searched  
`key` The value to be searched for  
**Returns:** The position of the search key, if it is contained in the array; otherwise,  $-index - 1$ , where `index` is the position where the element may be inserted

- **static `T[] copyOf(T[] a, int newLength)`**

This method copies the elements of the array `a`, or the first `newLength` elements if `a.length > newLength`, into an array of length `newLength` and returns that array. `T` can be a primitive type, class, or interface type.

**Parameters:** `a` The array to be copied  
`key` The value to be searched for  
**Returns:** The position of the search key, if it is contained in the array; otherwise,  $-index - 1$ , where `index` is the position where the element may be inserted

- **static void sort(`Object[] a`)**

This method sorts the specified array of objects into ascending order. Its elements must implement the `Comparable` interface.

**Parameters:** `a` The array to be sorted

- **static String toString(`T[] a`)**

This method creates and returns a string containing the array elements. `T` can be a primitive type, class, or interface type.

**Parameters:** `a` An array  
**Returns:** A string containing a comma-separated list of string representations of the array elements, surrounded by brackets.

## Class `java.util.Calendar`

- **int get(int field)**

This method returns the value of the given field.

**Parameters:** `field` One of `Calendar.YEAR`, `Calendar.MONTH`, `Calendar.DAY_OF_MONTH`, `Calendar.HOUR`, `Calendar.MINUTE`, `Calendar.SECOND`, or `Calendar.MILLISECOND`

## Interface `java.util.Collection<E>`

- **boolean add(`E element`)**

This method adds an element to this collection.

**Parameters:** `element` The element to add  
**Returns:** true if adding the element changes the collection

- **boolean contains(`E element`)**

This method tests whether an element is present in this collection.

**Parameters:** `element` The element to find  
**Returns:** true if the element is contained in the collection

- **Iterator iterator()**

This method returns an iterator that can be used to traverse the elements of this collection.

**Returns:** An object of a class implementing the `Iterator` interface

- **boolean remove(`E element`)**

This method removes an element from this collection.

**Parameters:** `element` The element to remove  
**Returns:** true if removing the element changes the collection

- **int size()**

This method returns the number of elements in this collection.

**Returns:** The number of elements in this collection

## Class `java.util.Collections`

- **static <T> int binarySearch(`List<T> a, T key`)**

This method searches the specified list for the specified object using the binary search algorithm. The list elements must implement the `Comparable` interface. The list must be sorted in ascending order.

**Parameters:** `a` The list to be searched  
`key` The value to be searched for  
**Returns:** The position of the search key, if it is contained in the list; otherwise,  $-index - 1$ , where `index` is the position where the element may be inserted

- **static <T> void sort(`List<T> a`)**

This method sorts the specified list of objects into ascending order. Its elements must implement the `Comparable` interface.

**Parameters:** `a` The list to be sorted

## Interface `java.util.Comparator<T>`

- **int compare(`T first, T second`)**

This method compares the given objects.

**Parameters:** `first, second` The objects to be compared

**Returns:** A negative integer if the first object is less than the second, zero if they are equal, or a positive integer otherwise

## A-22 Appendix D The Java Library

- static <T, U extends Comparable<? super U>>  
`Comparator<T> comparing(  
 Function<? super T, ? extends U> keyFunction)`

This method makes a comparator from a function that maps objects of type T to objects of a type U that implements the Comparable interface.

**Parameters:** keyFunction A function that derives a comparable key  
**Returns:** A comparator that compares the results of applying the key function. For example, `Comparator.comparing(account -> account.getBalance())` compares bank accounts by their balance.

### Class java.util.Date

- `Date()`

This constructs an object that represents the current date and time.

### Class java.util.EventObject

- `Object getSource()`

This method returns a reference to the object on which this event initially occurred.

**Returns:** The source of this event

### Class java.util.GregorianCalendar

- `GregorianCalendar()`

This constructs a calendar object that represents the current date and time.

- `GregorianCalendar(int year, int month, int day)`

This constructs a calendar object that represents the start of the given date.

**Parameters:** year, month, day The given date

### Class java.util.HashMap<K, V>

- `HashMap<K, V>()`

This constructs an empty hash map.

### Class java.util.HashSet<E>

- `HashSet<E>()`

This constructs an empty hash set.

### Class java.util.InputMismatchException

This exception is thrown if the next available input item does not match the type of the requested item.

### Interface java.util.Iterator<E>

- `boolean hasNext()`

This method checks whether the iterator is past the end of the list.

**Returns:** true if the iterator is not yet past the end of the list

- `E next()`

This method moves the iterator over the next element in the linked list. This method throws an exception if the iterator is past the end of the list.

**Returns:** The object that was just skipped over

- `void remove()`

This method removes the element that was returned by the last call to next or previous. This method throws an exception if there was an add or remove operation after the last call to next or previous.

### Class java.util.LinkedList<E>

- `void addFirst(E element)`

- `void addLast(E element)`

These methods add an element before the first or after the last element in this list.

**Parameters:** element The element to be added

- `E getFirst()`

- `E getLast()`

These methods return a reference to the specified element from this list.

**Returns:** The first or last element

- `E removeFirst()`

- `E removeLast()`

These methods remove the specified element from this list.

**Returns:** A reference to the removed element

### Interface java.util.List<E>

- `ListIterator<E> listIterator()`

This method gets an iterator to visit the elements in this list.

**Returns:** An iterator that points before the first element in this list

### Interface java.util.ListIterator<E>

Objects implementing this interface are created by the listIterator methods of list classes.

- `void add(E element)`

This method adds an element after the iterator position and moves the iterator after the new element.

**Parameters:** element The element to be added

- **boolean hasPrevious()**

This method checks whether the iterator is before the first element of the list.

**Returns:** true if the iterator is not before the first element of the list

- **E previous()**

This method moves the iterator over the previous element in the linked list. This method throws an exception if the iterator is before the first element of the list.

**Returns:** The object that was just skipped over

- **void set(E element)**

This method replaces the element that was returned by the last call to next or previous. This method throws an exception if there was an add or remove operation after the last call to next or previous.

**Parameters:** element The element that replaces the old list element

## Interface java.util.Map<K, V>

- **V get(K key)**

Gets the value associated with a key in this map.

**Parameters:** key The key for which to find the associated value

**Returns:** The value associated with the key, or null if the key is not present in the map

- **Set<K> keySet()**

This method returns all keys this map.

**Returns:** A set of all keys in this map

- **V put(K key, V value)**

This method associates a value with a key in this map.

**Parameters:** key The lookup key  
value The value to associate with the key

**Returns:** The value previously associated with the key, or null if the key was not present in the map

- **V remove(K key)**

This method removes a key and its associated value from this map.

**Parameters:** key The lookup key

**Returns:** The value previously associated with the key, or null if the key was not present in the map

## Class java.util.NoSuchElementException

This exception is thrown if an attempt is made to retrieve a value that does not exist.

## Class java.util.Objects

- **static int hashCode(Object... values)**

This method computes a hash code from a sequence of values.

**Parameters:** values A sequence of values

**Returns:** A hash code that combines the hash codes of the given values

## Class java.util.PriorityQueue<E>

- **PriorityQueue<E>()**

This constructs an empty priority queue. The element type E must implement the Comparable interface.

- **E remove()**

This method removes the smallest element in the priority queue.

**Returns:** The removed value

## Interface java.util.Queue<E>

- **E peek()**

Gets the element at the head of the queue without removing it.

**Returns:** The head element or null if the queue is empty

## Class java.util.Random

- **Random()**

This constructs a new random number generator.

- **double nextDouble()**

This method returns the next pseudorandom, uniformly distributed floating-point number between 0.0 (inclusive) and 1.0 (exclusive) from this random number generator's sequence.

**Returns:** The next pseudorandom floating-point number

- **int nextInt(int n)**

This method returns the next pseudorandom, uniformly distributed integer between 0 (inclusive) and the specified value (exclusive) drawn from this random number generator's sequence.

**Parameters:** n Number of values to draw from

**Returns:** The next pseudorandom integer

## A-24 Appendix D The Java Library

### Class java.util.Scanner

- `Scanner(File in)`
- `Scanner(InputStream in)`
- `Scanner(Reader in)`

These construct a scanner that reads from the given file, input stream, or reader.

**Parameters:** `in` The file, input stream, or reader from which to read

- `void close()`

This method closes this scanner and releases any associated system resources.

- `boolean hasNext()`
- `boolean hasNextDouble()`
- `boolean hasNextInt()`
- `boolean hasNextLine()`

These methods test whether it is possible to read any non-empty string, a floating-point value, an integer, or a line, as the next item.

**Returns:** true if it is possible to read an item of the requested type, false otherwise (either because the end of the file has been reached, or because a number type was tested and the next item is not the desired number type)

- `String next()`
- `double nextDouble()`
- `int nextInt()`
- `String nextLine()`

These methods read the next whitespace-delimited string, floating-point value, integer, or line.

**Returns:** The value that was read

- `Scanner useDelimiter(String pattern)`

Sets the pattern for the delimiters between input tokens.

**Parameters:** `pattern` A regular expression for the delimiter pattern

**Returns:** This scanner

### Interface java.util.Set<E>

This interface describes a collection that contains no duplicate elements.

### Class java.util.TreeMap<K, V>

- `TreeMap<K, V>()`

This constructs an empty tree map. The iterator of a TreeMap visits the entries in sorted order.

### Class java.util.TreeSet<E>

- `TreeSet<E>()`

This constructs an empty tree set.

## Package java.util.logging

### Class java.util.logging.Level

- `static final int INFO`

This value indicates informational logging.

- `static final int OFF`

This value indicates logging of no messages.

### Class java.util.logging.Logger

- `static Logger getGlobal()`

This method gets the global logger. For Java 5 and 6, use `getLogger("global")` instead.

**Returns:** The global logger that, by default, displays messages with level `INFO` or a higher severity on the console.

- `void info(String message)`

This method logs an informational message.

**Parameters:** `message` The message to log

- `void setLevel(Level aLevel)`

This method sets the logging level. Logging messages with a lesser severity than the current level are ignored.

**Parameters:** `aLevel` The minimum level for logging messages

## Package javax.swing

### Class javax.swing.AbstractButton

- `void addActionListener(ActionListener listener)`

This method adds an action listener to the button.

**Parameters:** `listener` The action listener to be added

- `boolean isSelected()`

This method returns the selection state of the button.

**Returns:** true if the button is selected

- `void setSelected(boolean state)`

This method sets the selection state of the button. This method updates the button but does not trigger an action event.

**Parameters:** `state` true to select, false to deselect

### Class javax.swing.ButtonGroup

- `void add(AbstractButton button)`

This method adds the button to the group.

**Parameters:** `button` The button to add

**Class javax.swing.ImageIcon**

- **`ImageIcon(String filename)`**

This constructs an image icon from the specified graphics file.

**Parameters:** filename A string specifying a file name

**Class javax.swing.JButton**

- **`JButton(String label)`**

This constructs a button with the given label.

**Parameters:** label The button label

**Class javax.swing.JCheckBox**

- **`JCheckBox(String text)`**

This constructs a check box with the given text, which is initially deselected. (Use the `setSelected` method to make the box selected; see the `javax.swing.AbstractButton` class.)

**Parameters:** text The text displayed next to the check box

**Class javax.swing.JComboBox**

- **`JComboBox()`**

This constructs a combo box with no items.

- **`void addItem(Object item)`**

This method adds an item to the item list of this combo box.

**Parameters:** item The item to add

- **`Object getSelectedItem()`**

This method gets the currently selected item of this combo box.

**Returns:** The currently selected item

- **`boolean isEditable()`**

This method checks whether the combo box is editable. An editable combo box allows the user to type into the text field of the combo box.

**Returns:** true if the combo box is editable

- **`void setEditable(boolean state)`**

This method is used to make the combo box editable or not.

**Parameters:** state true to make editable, false to disable editing

- **`void setSelectedItem(Object item)`**

This method sets the item that is shown in the display area of the combo box as selected.

**Parameters:** item The item to be displayed as selected

**Class javax.swing.JComponent**

- **`protected void paintComponent(Graphics g)`**

Override this method to paint the surface of a component. Your method needs to call `super.paintComponent(g)`.

**Parameters:** g The graphics context used for drawing

- **`void setBorder(Border b)`**

This method sets the border of this component.

**Parameters:** b The border to surround this component

- **`void setFont(Font f)`**

Sets the font used for the text in this component.

**Parameters:** f A font

**Class javax.swing.JFileChooser**

- **`JFileChooser()`**

This constructs a file chooser.

- **`File getSelectedFile()`**

This method gets the selected file from this file chooser.

**Returns:** The selected file

- **`int showOpenDialog(Component parent)`**

This method displays an “Open File” file chooser dialog box.

**Parameters:** parent The parent component or null

**Returns:** The return state of this file chooser after it has been closed by the user: either APPROVE\_OPTION or CANCEL\_OPTION. If APPROVE\_OPTION is returned, call `getSelectedFile()` on this file chooser to get the file

- **`int showSaveDialog(Component parent)`**

This method displays a “Save File” file chooser dialog box.

**Parameters:** parent The parent component or null

**Returns:** The return state of the file chooser after it has been closed by the user: either APPROVE\_OPTION or CANCEL\_OPTION

**Class javax.swing.JFrame**

- **`void setDefaultCloseOperation(int operation)`**

This method sets the default action for closing the frame.

**Parameters:** operation The desired close operation.

Choose among DO\_NOTHING\_ON\_CLOSE, HIDE\_ON\_CLOSE (the default), DISPOSE\_ON\_CLOSE, or EXIT\_ON\_CLOSE

## A-26 Appendix D The Java Library

### • void `setMenuBar(JMenuBar mb)`

This method sets the menu bar for this frame.

**Parameters:** `mb` The menu bar. If `mb` is `null`, then the current menu bar is removed

### • static final int `EXIT_ON_CLOSE`

This value indicates that when the user closes this frame, the application is to exit.

## Class `javax.swing.JLabel`

### • `JLabel(String text)`

### • `JLabel(String text, int alignment)`

These containers create a `JLabel` instance with the specified text and horizontal alignment.

**Parameters:** `text` The label text to be displayed by the label  
`alignment` One of `SwingConstants.LEFT`, `SwingConstants.CENTER`, or `SwingConstants.RIGHT`

## Class `javax.swing.JMenu`

### • `JMenu()`

This constructs a menu with no items.

### • `JMenuItem add(JMenuItem menuItem)`

This method appends a menu item to the end of this menu.

**Parameters:** `menuItem` The menu item to be added  
**Returns:** The menu item that was added

## Class `javax.swing.JMenuBar`

### • `JMenuBar()`

This constructs a menu bar with no menus.

### • `JMenu add(JMenu menu)`

This method appends a menu to the end of this menu bar.

**Parameters:** `menu` The menu to be added  
**Returns:** The menu that was added

## Class `javax.swing.JMenuItem`

### • `JMenuItem(String text)`

This constructs a menu item.

**Parameters:** `text` The text to appear in the menu item

## Class `javax.swing.JOptionPane`

### • static String `showInputDialog(Object prompt)`

This method brings up a modal input dialog box, which displays a prompt and waits for the user to enter an input in a text field, preventing the user from doing anything else in this program.

**Parameters:** `prompt` The prompt to display

**Returns:** The string that the user typed

### • static void `showMessageDialog(Component parent, Object message)`

This method brings up a confirmation dialog box that displays a message and waits for the user to confirm it.

**Parameters:** `parent` The parent component or `null`  
`message` The message to display

## Class `javax.swing.JPanel`

This class is a component without decorations. It can be used as an invisible container for other components.

## Class `javax.swing.JRadioButton`

### • `JRadioButton(String text)`

This constructs a radio button having the given text that is initially deselected. (Use the  `setSelected` method to select it; see the `javax.swing.AbstractButton` class.)

**Parameters:** `text` The string displayed next to the radio button

## Class `javax.swing.JScrollPane`

### • `JScrollPane(Component c)`

This constructs a scroll pane around the given component.

**Parameters:** `c` The component that is decorated with scroll bars

## Class `javax.swing.JSlider`

### • `JSlider(int min, int max, int value)`

This constructor creates a horizontal slider using the specified minimum, maximum, and value.

**Parameters:** `min` The smallest possible slider value  
`max` The largest possible slider value  
`value` The initial value of the slider

### • void `addChangeListener(ChangeListener listener)`

This method adds a change listener to the slider.

**Parameters:** `listener` The change listener to add

### • int `getValue()`

This method returns the slider's value.

**Returns:** The current value of the slider

## Class `javax.swing.JTextArea`

### • `JTextArea()`

This constructs an empty text area.

- **JTextArea(int rows, int columns)**

This constructs an empty text area with the specified number of rows and columns.

**Parameters:** rows The number of rows  
columns The number of columns

- **void append(String text)**

This method appends text to this text area.

**Parameters:** text The text to append

### Class javax.swing.JTextField

- **JTextField()**

This constructs an empty text field.

- **JTextField(int columns)**

This constructs an empty text field with the specified number of columns.

**Parameters:** columns The number of columns

### Class javax.swing.KeyStroke

- **static KeyStroke getKeyStrokeForEvent(KeyEvent event)**

Gets a KeyStroke object describing the key stroke that caused the event.

**Parameters:** event The key event to be analyzed

**Returns:** A KeyStroke object. Call `toString` on this object to get a string representation such as "pressed LEFT"

### Class javax.swing.Timer

- **Timer(int millis, ActionListener listener)**

This constructs a timer that notifies an action listener whenever a time interval has elapsed.

**Parameters:** millis The number of milliseconds between timer notifications  
listener The object to be notified when the time interval has elapsed

- **void start()**

This method starts the timer. Once the timer has started, it begins notifying its listener.

- **void stop()**

This method stops the timer. Once the timer has stopped, it no longer notifies its listener.

## Package javax.swing.border

### Class javax.swing.border.EtchedBorder

- **EtchedBorder()**

This constructor creates a lowered etched border.

### Class javax.swing.border.TitledBorder

- **TitledBorder(Border b, String title)**

This constructor creates a titled border that adds a title to a given border.

**Parameters:** b The border to which the title is added  
title The title the border should display

## Package javax.swing.event

### Class javax.swing.event.ChangeEvent

Components such as sliders emit change events when they are manipulated by the user.

### Interface javax.swing.event.ChangeListener

- **void stateChanged(ChangeEvent e)**

This event is called when the event source has changed its state.

**Parameters:** e A change event

## Package javax.swing.text

### Class javax.swing.text.JTextComponent

- **String getText()**

This method returns the text contained in this text component.

**Returns:** The text

- **boolean isEditable()**

This method checks whether this text component is editable.

**Returns:** true if the component is editable

- **void setEditable(boolean state)**

This method is used to make this text component editable or not.

**Parameters:** state true to make editable, false to disable editing

- **void setText(String text)**

This method sets the text of this text component to the specified text. If the argument is the empty string, the old text is deleted.

**Parameters:** text The new text to be set



# JAVA LANGUAGE CODING GUIDELINES

## Introduction

This coding style guide is a simplified version of one that has been used with good success both in industrial practice and for college courses.

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read code from your instructor and classmates. You will really appreciate that if you do a team project. It is also easier for your instructor and your grader to grasp the essence of your programs quickly.

A style guide makes you a more productive programmer because it reduces gratuitous choice. If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems.

In these guidelines, several constructs are plainly outlawed. That doesn't mean that programmers using them are evil or incompetent. It does mean that the constructs are not essential and can be expressed just as well or even better with other language constructs.

If you already have programming experience, in Java or another language, you may be initially uncomfortable at giving up some fond habits. However, it is a sign of professionalism to set aside personal preferences in minor matters and to compromise for the benefit of your group.

These guidelines are necessarily somewhat dull. They also mention features that you may not yet have seen in class. Here are the most important highlights:

- Tabs are set every three spaces.
- Variable and method names are lowercase, with occasional uppercase characters in the middle.
- Class names start with an Uppercase letter.
- Constant names are uppercase, with an occasional UNDER\_SCORE.
- There are spaces after reserved words and surrounding binary operators.
- Braces must line up horizontally or vertically.
- No magic numbers may be used.
- Every method, except for `main` and overridden methods, must have a comment.
- At most 30 lines of code may be used per method.
- No `continue` or `break` is allowed.
- All non-final variables must be private.

Note to the instructor: Of course, many programmers and organizations have strong feelings about coding style. If this style guide is incompatible with your own preferences or with local custom, please feel free to modify it.

# Source Files

Each Java program is a collection of one or more source files. The executable program is obtained by compiling these files. Organize the material in each file as follows:

- package statement, if appropriate
- import statements
- A comment explaining the purpose of this file
- A public class
- Other classes, if appropriate

The comment explaining the purpose of this file should be in the format recognized by the javadoc utility. Start with a `/**`, and use the `@author` and `@version` tags:

```
/**
 * Classes to manipulate widgets.
 * Solves CS101 homework assignment #3
 * COPYRIGHT (C) 2016 Harry Morgan. All Rights Reserved.
 * @author Harry Morgan
 * @version 1.01 2016-02-15
 */
```

# Classes

Each class should be preceded by a class comment explaining the purpose of the class.

First list all public features, then all private features.

Within the public and private sections, use the following order:

1. Instance variables
2. Static variables
3. Constructors
4. Instance methods
5. Static methods
6. Inner classes

Leave a blank line after every method.

All non-final variables must be private. (However, instance variables of a private inner class may be public.) Methods and final variables can be either public or private, as appropriate.

All features must be tagged `public` or `private`. Do not use the default visibility (that is, package visibility) or the `protected` attribute.

Avoid static variables (except final ones) whenever possible. In the rare instance that you need static variables, you are permitted one static variable per class.

# Methods

Every method (except for `main`) starts with a comment in javadoc format.

```
/**
 * Convert calendar date into Julian day.
 * Note: This algorithm is from Press et al., Numerical Recipes
 * in C, 2nd ed., Cambridge University Press, 1992.
 * @param day day of the date to be converted
 * @param month month of the date to be converted
 * @param year year of the date to be converted
 * @return the Julian day number that begins at noon of the
 * given calendar date.
 */
public static int getJulianDayNumber(int day, int month, int year)
{
 ...
}
```

Parameter variable names must be explicit, especially if they are integers or Boolean:

```
public Employee remove(int d, double s)
 // Huh?
public Employee remove(int department, double severancePay)
 // OK
```

Methods must have at most 30 lines of code. The method signature, comments, blank lines, and lines containing only braces are not included in this count. This rule forces you to break up complex computations into separate methods.

# Variables and Constants

Do not define all variables at the beginning of a block:

```
{
 double xold; // Don't
 double xnew;
 boolean done;
 ...
}
```

Define each variable just before it is used for the first time:

```
{
 ...
 double xold = Integer.parseInt(input);
 boolean done = false;
 while (!done)
 {
 double xnew = (xold + a / xold) / 2;
 ...
 }
 ...
}
```

Do not define two variables on the same line:

```
int dimes = 0, nickels = 0; // Don't
```

Instead, use two separate definitions:

```
int dimes = 0; // OK
int nickels = 0;
```

In Java, constants must be defined with the reserved word `final`. If the constant is used by multiple methods, declare it as `static final`. It is a good idea to define static final variables as private if no other class has an interest in them.

Do not use magic numbers! A magic number is a numeric constant embedded in code, without a constant definition. Any number except -1, 0, 1, and 2 is considered magic:

```
if (p.getX() < 300) // Don't
```

Use `final` variables instead:

```
final double WINDOW_WIDTH = 300;
...
if (p.getX() < WINDOW_WIDTH) // OK
```

Even the most reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant

```
public static final int DAYS_PER_YEAR = 365;
```

so that you can easily produce a Martian version without trying to find all the 365s, 364s, 366s, 367s, and so on, in your code.

When declaring array variables, group the `[]` with the type, not the variable.

```
int[] values; // OK
int values[]; // Ugh—this is an ugly holdover from C
```

When using collections, use type parameters and not “raw” types.

```
ArrayList<String> names = new ArrayList<>(); // OK
ArrayList names = new ArrayList(); // Not OK
```

## Control Flow

### Statement Bodies

Use braces to enclose the bodies of branch and loop statements, even if they contain only a single statement. For example,

```
if (x < 0)
{
 x++;
}
```

and not

```
if (x < 0)
 x++; // Not OK--no braces
```

### The for Statement

Use for loops only when a variable runs from somewhere to somewhere with some constant increment/decrement:

```
for (int i = 0; i < a.length; i++)
{
```

```
 System.out.println(a[i]);
}
```

Or, even better, use the enhanced for loop:

```
for (int e : a)
{
 System.out.println(e);
}
```

Do not use the for loop for weird constructs such as

```
for (a = a / 2; count < ITERATIONS; System.out.println(xnew)) // Don't
```

Make such a loop into a while loop. That way, the sequence of instructions is much clearer:

```
a = a / 2;
while (count < ITERATIONS) // OK
{
 ...
 System.out.println(xnew);
}
```

## Nonlinear Control Flow

Avoid the switch statement, because it is easy to fall through accidentally to an unwanted case. Use if/else instead.

Avoid the break or continue statements. Use another boolean variable to control the execution flow.

## Exceptions

Do not tag a method with an overly general exception specification:

```
Widget readWidget(Reader in) throws Exception // Bad
```

Instead, specifically declare any checked exceptions that your method may throw:

```
Widget readWidget(Reader in)
 throws IOException, MalformedWidgetException // Good
```

Do not “squelch” exceptions:

```
try
{
 double price = in.readDouble();
}
catch (Exception e)
{ } // Bad
```

Beginners often make this mistake “to keep the compiler happy”. If the current method is not appropriate for handling the exception, simply use a throws specification and let one of its callers handle it.

Always use the try-with-resources statement to ensure that resources are closed even when an exception occurs. For example,

```
try (Scanner in = new Scanner(. . .); PrintWriter out = new PrintWriter(. . .))
{
 while (in.hasNextLine())
 {
 out.println(in.nextLine());
 }
}
```

# Lexical Issues

## Naming Conventions

The following rules specify when to use upper- and lowercase letters in identifier names:

- All variable and method names are in lowercase (maybe with an occasional uppercase in the middle); for example, `firstPlayer`.
- All constants are in uppercase (maybe with an occasional `UNDER_SCORE`); for example, `CLOCK_RADIUS`.
- All class and interface names start with uppercase and are followed by lowercase letters (maybe with an occasional `UpperCase` letter); for example, `BankTeller`.
- Generic type variables are in uppercase, usually a single letter.

Names must be reasonably long and descriptive. Use `firstPlayer` instead of `fp`. No drppng f vwls. Local variables that are fairly routine can be short (`ch`, `i`) as long as they are really just boring holders for an input character, a loop counter, and so on. Also, do not use `ctr`, `c`, `cntr`, `cnt`, `c2` for variables in your method. Surely these variables all have specific purposes and can be named to remind the reader of them (for example, `current`, `next`, `previous`, `result`, ...). However, it is customary to use single-letter names, such as `T` or `E` for generic types.

## Indentation and White Space

Use tab stops every three columns. That means you will need to change the tab stop setting in your editor!

Use blank lines freely to separate parts of a method that are logically distinct.

Use a blank space around every binary operator:

```
x1 = (-b - Math.sqrt(b * b - 4 * a * c)) / (2 * a);
// Good
```

```
x1=(-b-Math.sqrt(b*b-4*a*c))/(2*a);
// Bad
```

Leave a blank space after (and not before) each comma or semicolon. Do not leave a space before or after a parenthesis or bracket in an expression. Leave spaces around the `( . . . )` part of an `if`, `while`, `for`, or `catch` statement.

```
if (x == 0) { y = 0; }

f(a, b[i]);
```

Every line must fit in 80 columns. If you must break a statement, add an indentation level for the continuation:

```
a[n] =;
+;
```

Start the indented line with an operator (if possible).

## Braces

Opening and closing braces must line up, either horizontally or vertically:

```
while (i < n) { System.out.println(a[i]); i++; }
```

```

while (i < n)
{
 System.out.println(a[i]);
 i++;
}

```

Some programmers don't line up vertical braces but place the { behind the reserved word:

```

while (i < n) { // DON'T
 System.out.println(a[i]);
 i++;
}

```

Doing so makes it hard to check that the braces match.

## Unstable Layout

Some programmers take great pride in lining up certain columns in their code:

```

firstRecord = other.firstRecord;
lastRecord = other.lastRecord;
cutoff = other.cutoff;

```

This is undeniably neat, but the layout is not stable under change. A new variable name that is longer than the preallotted number of columns requires that you move all entries around:

```

firstRecord = other.firstRecord;
lastRecord = other.lastRecord;
cutoff = other.cutoff;
marginalFudgeFactor = other.marginalFudgeFactor;

```

This is just the kind of trap that makes you decide to use a short variable name like `mff` instead. Use a simple layout that is easy to maintain as your programs change.

# TOOL SUMMARY

In this summary, we use a monospaced font for actual commands such as `javac`. An italic font denotes descriptions of tool command components such as *options*. Items enclosed in brackets [...] are optional. Items separated by vertical bars | are alternatives. Do not include the brackets or vertical bars when typing the commands.

## The Java Compiler

`javac [options] sourceFile1|@fileList1 sourceFile2|@fileList2 . . .`

A file list is a text file that contains one file name per line. For example,

`Greeting.list`

```
1 Greeting.java
2 GreetingTester.java
```

Then you can compile all files with the command

`javac @Greeting.list`

The Java compiler options are summarized in Table 1.

**Table 1** Common Compiler Options

Option	Description
<code>-classpath locations</code> or <code>-cp locations</code>	The compiler is to look for classes on this path, overriding the CLASSPATH environment variable. If neither is specified, the current directory is used. Each <i>location</i> is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator ( : on Unix, ; on Windows).
<code>-sourcepath locations</code>	The compiler is to look for source files on this path. If not specified, source files are searched in the class path.
<code>-d directory</code>	The compiler places files into the specified directory.
<code>-g</code>	Generate debugging information.
<code>-verbose</code>	Include information about all classes that are being compiled (useful for troubleshooting).
<code>-deprecation</code>	Give detailed information about the usage of deprecated messages.
<code>-Xlint:errorType</code>	Carry out additional error checking. If you get warnings about unchecked conversions, compile with the <code>-Xlint:unchecked</code> option.

# The Java Virtual Machine Launcher

The following command loads the given class and starts its `main` method, passing it an array containing the provided command line arguments:

```
java [options] ClassName [argument1 argument2 . . .]
```

The following command loads the main class of the given JAR file and starts its `main` method, passing it an array containing the provided command line arguments:

```
java [options] -jar jarFileName [argument1 argument2 . . .]
```

The Java virtual machine options are summarized in Table 2.

**Table 2 Common Virtual Machine Launcher Options**

Option	Description
<code>-classpath</code> <i>locations</i> or <code>-cp</code> <i>locations</i>	Look for classes on this path, overriding the <code>CLASSPATH</code> environment variable. If neither is specified, the current directory is used. Each <i>location</i> is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: on Unix, ; on Windows).
<code>-verbose</code>	Trace class loading
<code>-Dproperty=value</code>	Set a system property that you can retrieve with the <code>System.getProperties</code> method.

# The JAR Tool

To combine one or more files into a JAR (Java Archive) file, use the command

```
jar cvf jarFile file1 file2 . . .
```

The resulting JAR file can be included in a class path.

To build a program that can be launched with `java -jar`, you must create a *manifest file*, such as

**myprog.mf**

**1 Main-Class: com/horstmann/MyProg**

The manifest must specify the path name of the class file that launches the application, but with the `.class` extension removed. Then build the JAR file as

```
jar cvfm jarFile manifestFile file1 file2 . . .
```

You can also use JAR as a replacement for a ZIP utility, simply to compress and bundle a set of files for any purpose. Then you may want to suppress the generation of the JAR manifest, with the command

```
jar cvfM jarFile file1 file2 . . .
```

To extract the contents of a JAR file into the current directory, use

```
jar xvf jarFile
```

To see the files contained in a JAR file without extracting the files, use

```
jar tvf jarFile
```

## The javadoc Tool

To extract documentation comments (summarized in the following section), run the javadoc program:

```
javadoc [options] sourceFile1|packageName1@fileList1
sourceFile2|packageName2@fileList2 . . .
```

Commonly used options are summarized in Table 3. See the documentation of the javac command in the first section of this appendix for an explanation of file lists.

To document all files in the current directory, use (all on one line)

```
javadoc -link http://download.oracle.com/javase/8/docs/api -d docdir *.java
```

**Table 3** Common javadoc Command Line Options

Option	Description
<code>-link URL</code>	Link to another set of javadoc files. You should include a link to the standard library documentation, either locally or at <a href="http://download.oracle.com/javase/8/docs/api">http://download.oracle.com/javase/8/docs/api</a> .
<code>-d directory</code>	Store the output in <i>directory</i> . This is a useful option, because it keeps your current directory from being cluttered up with javadoc files.
<code>-classpath locations</code>	Look for classes on the specified paths, overriding the CLASSPATH environment variable. If neither is specified, the current directory is used. Each <i>location</i> is a directory, JAR file, or ZIP file. Locations are separated by a platform-dependent separator (: Unix, ; Windows).
<code>-sourcepath locations</code>	Look for source files on the specified paths. If not specified, source files are searched in the class path.
<code>-author, -version</code>	Include author, version information in the documentation. This information is omitted by default.

## Documentation Comments

A documentation comment is delimited by `/**` and `*/`. You can comment

- Classes
- Methods
- Instance variables

Each comment is placed *immediately above* the feature it documents.

Each `/** . . . */` documentation comment contains introductory text followed by tagged documentation. A tag starts with an @ character, such as `@author` or `@param`. Tags are summarized in Table 4. The *first sentence* of the introductory text should be a summary statement. The javadoc utility automatically generates summary pages that extract these sentences.

You can use HTML tags such as `em` for emphasis, `code` for a monospaced font, `img` for images, `ul` for bulleted lists, and so on.

Here is a typical example. The summary sentence (in color) will be included with the method summary.

```
/***
 Withdraws money from the bank account. Increments the
 transaction count.
 @param amount the amount to withdraw
 @return the balance after the withdrawal
 @throws IllegalArgumentException if the balance is not sufficient
*/
public double withdraw(double amount)
{
 if (balance - amount < minimumBalance)
 {
 throw new IllegalArgumentException();
 }
 balance = balance - amount;
 transactions++;
 return balance;
}
```

**Table 4** Common javadoc Tags

Tag	Description
<code>@param parameter explanation</code>	A parameter of a method. Use a separate tag for each parameter.
<code>@return explanation</code>	The return value of a method.
<code>@throws exceptionType explanation</code>	An exception that a method may throw. Use a separate tag for each exception.
<code>@deprecated</code>	A feature that remains for compatibility but that should not be used for new code.
<code>@see packageName.ClassName</code> <code>@see packageName.ClassName</code> <code>#methodName(Type<sub>1</sub>, Type<sub>2</sub>, . . .)</code> <code>@see packageName.ClassName#variableName</code>	A reference to a related documentation entry.
<code>@author</code>	The author of a class or interface. Use a separate tag for each author.
<code>@version</code>	The version of a class or interface.

# NUMBER SYSTEMS

## Binary Numbers

Decimal notation represents numbers as powers of 10, for example

$$1729_{\text{decimal}} = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

There is no particular reason for the choice of 10, except that several historical number systems were derived from people's counting with their fingers. Other number systems, using a base of 12, 20, or 60, have been used by various cultures throughout human history. However, computers use a number system with base 2 because it is far easier to build electronic components that work with two values, which can be represented by a current being either off or on, than it would be to represent 10 different values of electrical signals. A number written in base 2 is also called a *binary* number.

For example,

$$1101_{\text{binary}} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

For digits after the “decimal” point, use negative powers of 2.

$$\begin{aligned} 1.101_{\text{binary}} &= 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 + \frac{1}{2} + \frac{1}{8} \\ &= 1 + 0.5 + 0.125 = 1.625 \end{aligned}$$

In general, to convert a binary number into its decimal equivalent, simply evaluate the powers of 2 corresponding to digits with value 1, and add them up. Table 1 shows the first powers of 2.

To convert a decimal integer into its binary equivalent, keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the *last* one. For example,

$$\begin{aligned} 100 \div 2 &= 50 \text{ remainder } 0 \\ 50 \div 2 &= 25 \text{ remainder } 0 \\ 25 \div 2 &= 12 \text{ remainder } 1 \\ 12 \div 2 &= 6 \text{ remainder } 0 \\ 6 \div 2 &= 3 \text{ remainder } 0 \\ 3 \div 2 &= 1 \text{ remainder } 1 \\ 1 \div 2 &= 0 \text{ remainder } 1 \end{aligned}$$

Therefore,  $100_{\text{decimal}} = 1100100_{\text{binary}}$ .

Conversely, to convert a fractional number less than 1 to its binary format, keep multiplying by 2. If the result is greater than 1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the *first* one. For example,

$$0.35 \cdot 2 = 0.7$$

$$0.7 \cdot 2 = 1.4$$

$$0.4 \cdot 2 = 0.8$$

$$0.8 \cdot 2 = 1.6$$

$$0.6 \cdot 2 = 1.2$$

$$0.2 \cdot 2 = 0.4$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 ...

To convert any floating-point number into binary, convert the whole part and the fractional part separately.

**Table 1 Powers of Two**

Power	Decimal Value
$2^0$	1
$2^1$	2
$2^2$	4
$2^3$	8
$2^4$	16
$2^5$	32
$2^6$	64
$2^7$	128
$2^8$	256
$2^9$	512
$2^{10}$	1,024
$2^{11}$	2,048
$2^{12}$	4,096
$2^{13}$	8,192
$2^{14}$	16,384
$2^{15}$	32,768
$2^{16}$	65,536

# Overflow and Roundoff Errors

In Java, an `int` value is an integer that is 32 bits long. When combining two such values, it is possible that the result does not fit into 32 bits. In that case, only the last 32 bits of the results are used, yielding an incorrect answer. For example,

```
int fiftyMillion = 50000000;
System.out.println(100 * fiftyMillion); // Expected: 5000000000
```

displays 705032704.

To see why this curious value is the result, one can carry out the long multiplication by hand:

```

1 1 0 0 1 0 0 * 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0
 0
 0
 1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0
 0
 0
 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0

```

The result has 33 bits. However, you can't fit a 33-bit result into a 32-bit int, and the top bit is discarded. The last 32 bits are the binary representation of 705032704. (Note that the top bit is  $2^{32} = 4294967296$ , and the two values add up to 5000000000, the correct result.)

With floating-point numbers, you can encounter another type of error: roundoff error. Consider this example:

```
double price = 4.35;
double quantity = 100;
double total = price * quantity; // Should be 100 * 4.35 = 435
System.out.println(total); // Prints 434.9999999999999
```

To see why the error occurs, carry out the long multiplication:

That is, the result is 434, followed by an infinite number of 1s. The fractional part of the product is the binary equivalent of an infinite decimal fraction  $0.999999\dots$ , which is equal to 1. But the CPU can store only a finite number of 1s, and it discards some of them when converting the result to a decimal number.

## Two's Complement Integers

To represent negative integers, there are two common representations, called “signed magnitude” and “two’s complement”. Signed magnitude notation is simple: use the leftmost bit for the sign (0 = positive, 1 = negative). For example, when using 8-bit numbers,

$$-13 = 10001101_{\text{signed magnitude}}$$

However, building circuitry for adding numbers gets a bit more complicated when one has to take a sign bit into account. The two’s complement representation solves this problem. To form the two’s complement of a number,

- Flip all bits.
- Then add 1.

For example, to compute  $-13$  as an 8-bit value, first flip all bits of  $00001101$  to get  $11110010$ . Then add 1:

$$-13 = 11110011_{\text{two's complement}}$$

Now no special circuitry is required for adding two numbers. Simply follow the normal rule for addition, with a carry to the next position if the sum of the digits and the prior carry is 2 or 3. For example,

$$\begin{array}{r} & \overset{1}{1} \ 1111 \ 111 \\ +13 & 0000 \ 1101 \\ -13 & 1111 \ 0011 \\ \hline 1 & 0000 \ 0000 \end{array}$$

But only the last 8 bits count, so  $+13$  and  $-13$  add up to 0, as they should.

In particular,  $-1$  has two’s complement representation  $1111\dots1111$ , with all bits set.

The leftmost bit of a two’s complement number is 0 if the number is positive or zero, 1 if it is negative.

Two’s complement notation with a given number of bits can represent one more negative number than positive numbers. For example, the 8-bit two’s complement numbers range from  $-128$  to  $+127$ .

This phenomenon is an occasional cause for a programming error. For example, consider the following code:

```
short b = ...;
if (b < 0) { b = (byte) -b; }
```

This code does not guarantee that  $b$  is nonnegative afterwards. If  $b$  happens to be  $-128$ , then computing its negative again yields  $-128$ . (Try it out—take  $10000000$ , flip all bits, and add 1.)

# IEEE Floating-Point Numbers

The Institute for Electrical and Electronics Engineering (IEEE) defines standards for floating-point representations in the IEEE-754 standard. Figure 1 shows how single-precision (`float`) and double-precision (`double`) values are decomposed into

- A sign bit
- An exponent
- A mantissa

Floating-point numbers use scientific notation, in which a number is represented as

$$b_0.b_1b_2b_3\dots \times 2^e$$

In this representation,  $e$  is the exponent, and the digits  $b_0.b_1b_2b_3\dots$  form the mantissa. The *normalized* representation is the one where  $b_0 \neq 0$ . For example,

$$100_{\text{decimal}} = 1100100_{\text{binary}} = 1.100100 \times 2^6$$

In the binary number system, because the first bit of a normalized representation must be 1, it is not actually stored in the mantissa. Therefore, you always need to add it on to represent the actual value. For example, the mantissa 1.100100 is stored as 100100.

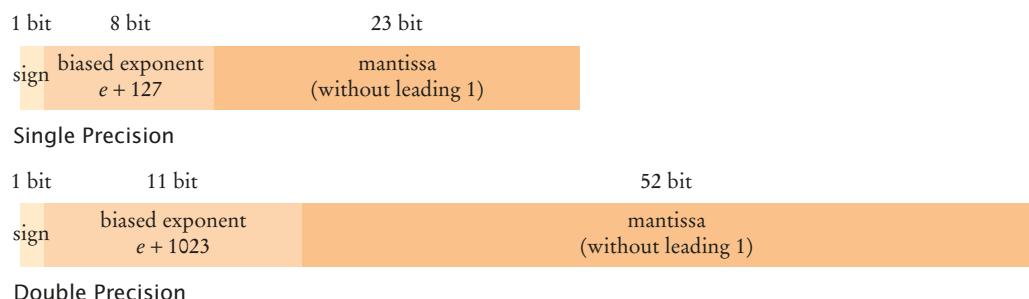
The exponent part of the IEEE representation uses neither signed magnitude nor two's complement representation. Instead, a *bias* is added to the actual exponent. The bias is 127 for single-precision numbers and 1023 for double-precision numbers. For example, the exponent  $e = 6$  would be stored as 133 in a single-precision number.

Thus,

$$100_{\text{decimal}} = 0\ 10000101\ 10010000000000000000000000000000 \text{ single-precision IEEE}$$

In addition, there are several special values. Among them are:

- *Zero*: biased exponent = 0, mantissa = 0.
- *Infinity*: biased exponent = 11...1, mantissa =  $\pm 0$ .
- *NaN* (not a number): biased exponent = 11...1, mantissa  $\neq \pm 0$ .



**Figure 1** IEEE Floating-Point Representation

# Hexadecimal Numbers

Because binary numbers can be hard to read for humans, programmers often use the hexadecimal number system, with base 16. The digits are denoted as 0, 1, ..., 9, A, B, C, D, E, F. (See Table 2.)

Four binary digits correspond to one hexadecimal digit. That makes it easy to convert between binary and hexadecimal values. For example,

$$11|1011|0001_{\text{binary}} = 3B1_{\text{hexadecimal}}$$

In Java, hexadecimal numbers are used for Unicode character values, such as \u03B1 (the Greek lowercase letter alpha). Hexadecimal integers are denoted with a 0x prefix, such as 0x3B1.

**Table 2** Hexadecimal Digits

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

## Bit and Shift Operations

There are four bit operations in Java: the unary negation (`~`) and the binary *and* (`&`), *or* (`|`), and *exclusive or* (`^`), often called *xor*.

Tables 3 and 4 show the truth tables for the bit operations in Java. When a bit operation is applied to integer values, the operation is carried out on corresponding bits.

For example, suppose you want to compute  $46 \& 13$ . First convert both values to binary.  $46_{\text{decimal}} = 101110_{\text{binary}}$  (actually `000000000000000000000000101110` as a 32-bit integer), and  $13_{\text{decimal}} = 1101_{\text{binary}}$ . Now combine corresponding bits:

$$\begin{array}{r} 0 \dots 0101110 \\ \& 0 \dots 0001101 \\ \hline 0 \dots 0001100 \end{array}$$

The answer is  $1100_{\text{binary}} = 12_{\text{decimal}}$ .

You sometimes see the `|` operator being used to combine two bit patterns. For example, the symbolic constant `BOLD` is the value 1, and the symbolic constant `ITALIC` is 2. The binary *or* combination `BOLD | ITALIC` has both the bold and the italic bit set:

$$\begin{array}{r} 0 \dots 0000001 \\ | 0 \dots 0000010 \\ \hline 0 \dots 0000011 \end{array}$$

Don't confuse the `&` and `|` bit operators with the `&&` and `||` operators. The latter work only on boolean values, not on bits of numbers.

**Table 3** The Unary Negation Operation

a	$\sim a$
0	1
1	0

**Table 4** The Binary And, Or, and Xor Operations

a	b	$a \& b$	$a   b$	$a ^ b$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Besides the operations that work on individual bits, there are three *shift* operations that take the bit pattern of a number and shift it to the left or right by a given number of positions. There are three shift operations: left shift (`<<`), right shift with sign extension (`>>`), and right shift with zero extension (`>>>`).

The left shift moves all bits to the left, filling in zeroes in the least significant bits. Shifting to the left by  $n$  bits yields the same result as multiplication by  $2^n$ . The right shift with sign extension moves all bits to the right, propagating the sign bit.

Therefore, the result is the same as integer division by  $2^n$ , both for positive and negative values. Finally, the right shift with zero extension moves all bits to the right, filling in zeroes in the most significant bits. (See Figure 2.)

Note that the right-hand-side value of the shift operators is reduced modulo 32 (for int values) or 64 (for long values) to determine the actual number of bits to shift.

For example,  $1 \ll 35$  is the same as  $1 \ll 3$ . Actually shifting 1 by 35 bits to the left would make no sense—the result would be 0.

The expression

$1 \ll n$

yields a bit pattern in which the  $n$ th bit is set (where the 0 bit is the least significant bit).

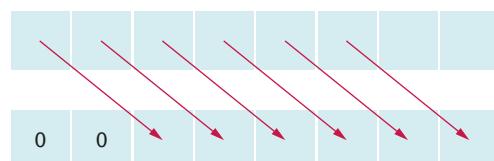
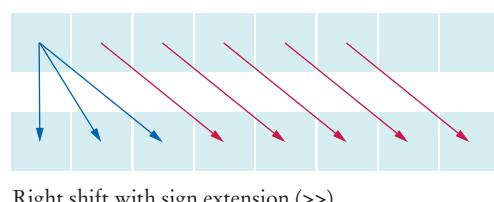
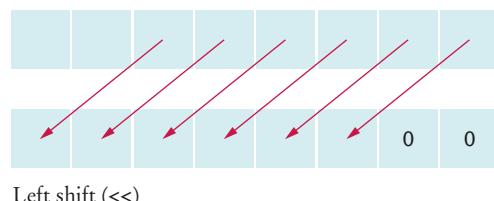
To set the  $n$ th bit of a number, carry out the operation

$x = x | 1 \ll n$

To check whether the  $n$ th bit is set, execute the test

`if ((x & 1 << n) != 0) . . .`

Note that the parentheses around the `&` are required—the `&` operator has a lower precedence than the relational operators.



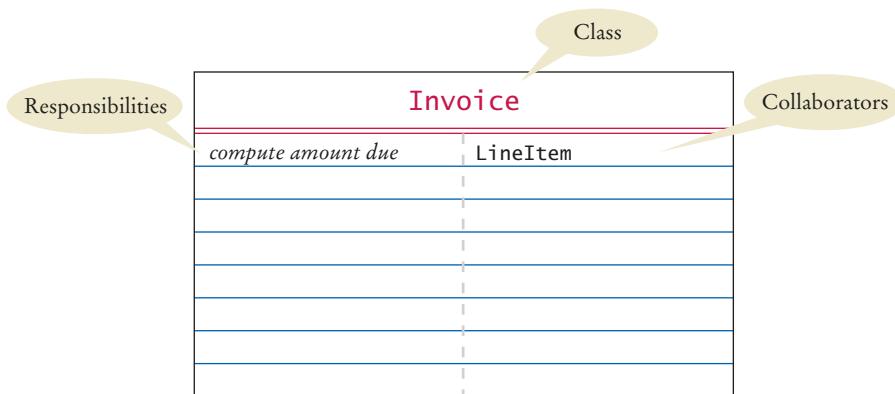
**Figure 2**  
The Shift Operations

# UML SUMMARY

In this book, we use a very restricted subset of the UML notation. This appendix lists the components of the subset.

## CRC Cards

CRC cards are used to describe in an informal fashion the responsibilities and collaborators for a class. Figure 1 shows a typical CRC card.

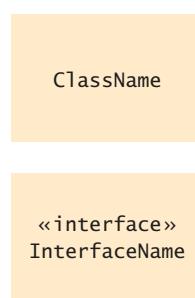


**Figure 1** Typical CRC Card

## UML Diagrams

Figure 2 shows the UML notation for classes and interfaces. You can optionally supply attributes and methods in a class diagram, as in Figure 3.

**Figure 2**  
UML Symbols for Classes  
and Interfaces



**Figure 3**  
Attributes and Methods  
in a Class Diagram

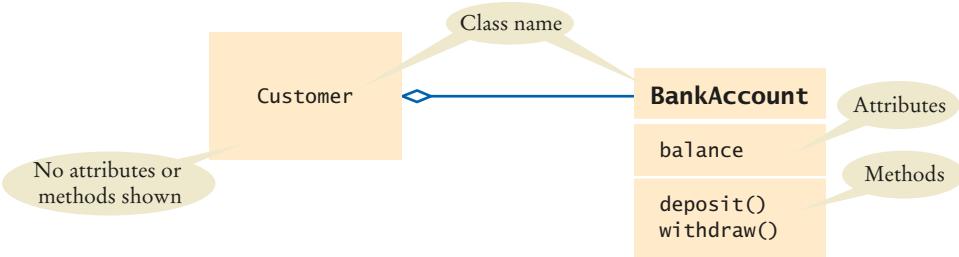


Table 1 shows the arrows used to indicate relationships between classes. Multiplicity can be indicated in a diagram, as in Figure 4.

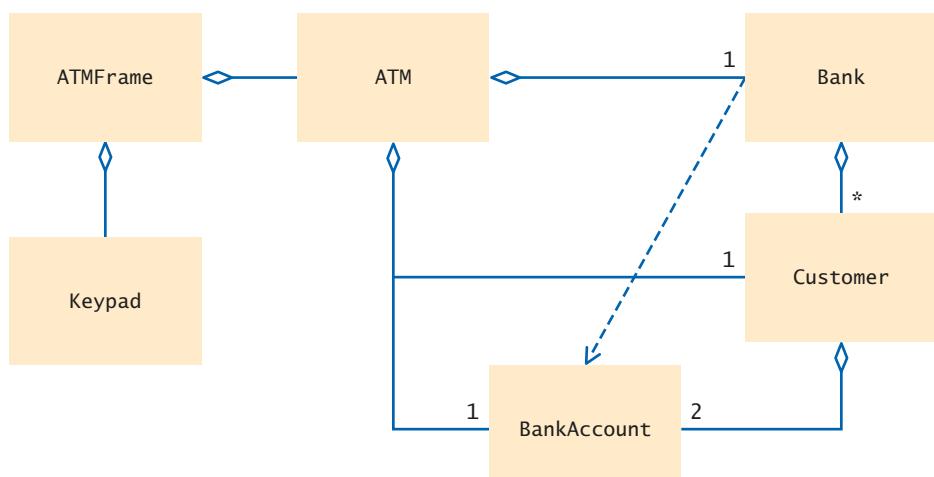
**Table 1 UML Relationship Symbols**

Relationship	Symbol	Line Style	Arrow Tip
Inheritance	—→	Solid	Triangle
Interface Implementation	- - - →	Dotted	Triangle
Aggregation	◊ —→	Solid	Diamond
Dependency	- - - - →	Dotted	Open

**Figure 4**  
An Aggregation Relationship  
with Multiplicities



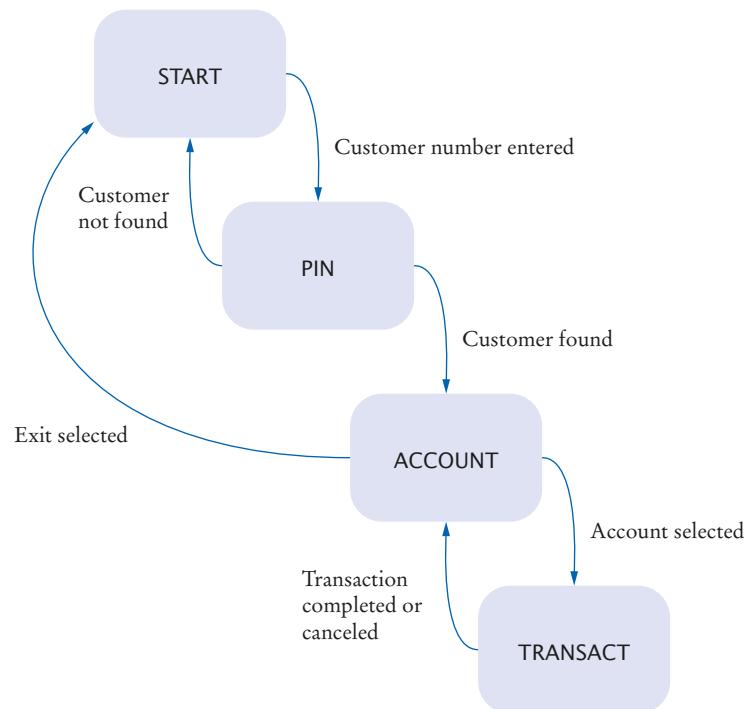
Dependencies between objects are described by a dependency diagram. Figure 5 is a typical example.



**Figure 5**  
UML Class Diagram for  
the ATM Simulation

State diagrams are used when an object goes through a discrete set of states that affects its behavior (see Figure 6).

For a complete discussion of the UML notation, see *The Unified Modeling Language User Guide*, by Booch, Rumbaugh, and Jacobson (Addison-Wesley, 2005).



**Figure 6** UML State Diagram for the ATM Class

# JAVA SYNTAX SUMMARY

In this syntax summary, we use a monospaced font for actual Java reserved words and tokens such as `while`. An italic font denotes language constructs such as *condition* or *variable*. Items enclosed in brackets [ ] are optional. Items separated by vertical bars | are alternatives. Do not include the brackets or vertical bars in your code!

The summary reflects the parts of the Java language that were covered in this book. For a full overview of the Java syntax, see <http://download.oracle.com/javase/8/docs/api/>.

Please be careful to distinguish an ellipsis . . . from the ... token. The latter appears twice in this appendix in the “variable parameters” discussion in the “Methods” section.

## Types

A type is a primitive type or a reference type. The primitive types are

- The numeric types `int`, `long`, `short`, `char`, `byte`, `float`, `double`
- The `boolean` type

The reference types are

- Classes such as `String` or `Employee`
- Enumeration types such as `enum Sex { FEMALE, MALE }`
- Interfaces such as `Comparable`
- Array types such as `Employee[]` or `int[][]`

## Variables

Local variable declarations have the form

`[final] Type variableName [= initializer];`

Examples:

```
int n;
double x = 0;
String harry = "Harry Handsome";
Rectangle box = new Rectangle(5, 10, 20, 30);
int[] a = { 1, 4, 9, 16, 25 };
```

The variable name consists only of letters, numbers, and underscores. It must begin with a letter or underscore. Names are case-sensitive: `totalscore`, `TOTALSCORE`, and `totalScore` are three different variables.

The scope of a local variable extends from the point of its definition to the end of the enclosing block.

A variable that is declared as `final` can have its value set only once.  
Instance variables will be discussed under “Classes”.

## Expressions

An *expression* is a variable, a method call, or a combination of subexpressions joined by operators. Examples are:

```
x
Math.sin(x)
x + Math.sin(x)
x * (1 + Math.sin(x))
x++
x == y
x == y && (z > 0 || w > 0)
p.x
e.getSalary()
v[i]
```

Operators can be *unary*, *binary*, or *ternary*. A unary operator acts on a single expression, such as `x++`. A binary operator combines two expressions, such as `x + y`. A ternary operator combines three expressions. Java has one ternary operator, `? :` (see Special Topic 5.1).

Unary operators can be *prefix* or *postfix*. A prefix operator is written before the expression on which it operates, as in `-x`. A postfix operator is written after the expression on which it operates, such as `x++`.

Operators are ranked by *precedence* levels. Operators with a higher precedence bind more strongly than operators with a lower precedence. For example, `*` has a higher precedence than `+`, so `x + y * z` is the same as `x + (y * z)`, even though the `+` comes first.

Most operators are *left-associative*. That is, operators of the same precedence are evaluated from the left to the right. For example, `x - y + z` is interpreted as `(x - y) + z`, not `x - (y + z)`. The exceptions are the unary prefix operators and the assignment operator which are right-associative. For example, `z = y = Math.sin(x)` means the same as `z = (y = Math.sin(x))`.

Appendix B has a list of all Java operators.

## Classes

The syntax for a *class* is

```
[public] [abstract|final] class ClassName
 [extends SuperClassName]
 [implements InterfaceName1, InterfaceName2, . . .]
{
 feature1
 feature2
 . . .
}
```

Each *feature* is either a declaration of the form

*modifiers constructor|method|instance variable|class*

or an initialization block

[static] { *body* }

See the section “Constructors” for more information about initialization blocks.

Potential *modifiers* include `public`, `private`, `protected`, `static`, and `final`.

An *instance variable* declaration has the form

*Type variableName [= initializer];*

A *constructor* has the form

```
ClassName(parameter1, parameter2, . . .)
 [throws ExceptionType1, ExceptionType2, . . .]
{
 body
}
```

A *method* has the form

```
Type methodName(parameter1, parameter2, . . .)
 [throws ExceptionType1, ExceptionType2, . . .]
{
 body
}
```

An *abstract method* has the form

`abstract Type methodName(parameter1, parameter2, . . .);`

Here is an example:

```
public class Point
{
 private double x; // Instance variable
 private double y;

 public Point() // Constructor with no arguments
 {
 x = 0; y = 0;
 }

 public Point(double xx, double yy) // Constructor
 {
 x = xx; y = yy;
 }

 public double getX() // Method
 {
 return x;
 }

 public double getY() // Method
 {
 return y;
 }
}
```

A class can have both instance variables and static variables. Each object of the class has a separate copy of the instance variables. There is only a one per-class copy of the static variables.

A class that is declared as `abstract` cannot be instantiated. That is, you cannot construct objects of that class.

A class that is declared as `final` cannot be extended.

## Interfaces

The syntax for an interface is

```
[public] interface InterfaceName
 [extends InterfaceName1, InterfaceName2, . . .]
{
 feature1
 feature2
 .
}
```

Each feature has the form

*modifiers method|variable*

Potential modifiers are `default`, `public`, `static`, `final`. Methods are automatically `public` and variables are automatically `public static final`. Default and static methods have method bodies.

A variable declaration has the form

*Type variableName = initializer;*

A method declaration has the form

*Type methodName(parameter<sub>1</sub>, parameter<sub>2</sub>, . . .);*

Here is an example:

```
public interface Measurable
{
 int CM_PER_INCH = 2.54;

 int getMeasure();

 static boolean isSmallerThan(Measurable other)
 {
 return getMeasure() < other.getMeasure();
 }
}
```

## Enumeration Types

The syntax for an enumeration type is

```
[public] enum EnumerationTypeName
{
 constant1, constant2, . . .;
 feature1
 feature2
 .
}
```

Each constant is a constant name, followed by optional construction parameters.

*constantName[(parameter<sub>1</sub>, parameter<sub>2</sub>, . . .)]*

The semicolon after the constants is only required if the enumeration declares additional features. An enumeration can have the same features as a class. Each feature has the form

*modifiers method|instance variable*

Potential modifiers are `public`, `static`, `final`.

Here are two examples:

```
public enum Suit { HEARTS, DIAMONDS, SPADES, CLUBS };
public enum Card
{
 TWO(2), THREE(3), FOUR(4), FIVE(5), SIX(6),
 SEVEN(7), EIGHT(8), NINE(9), TEN(10),
 JACK(10), QUEEN(10), KING(10), ACE(11);
 private int value;

 public void Card(int aValue) { value = aValue; }
 public int getValue() { return value; }
}
```

## Methods

A method definition has the form

```
modifiers Type methodName(parameter1, parameter2, . . . , parametern)
 [throws ExceptionType1, ExceptionType2, . . .]
{
 body
}
```

The return type *Type* is any Java type, or the special type `void` to indicate that the method returns no value.

Each *parameter variable* has the form

[final] *Type parameterName*

A method has variable parameters if the last parameter variable has the special form

*Type... parameterName*

Such a method can be called with a sequence of values of the given type of any length. The parameter variable with the given name is an array of the given type that holds the arguments. For example, the method

```
public static double sum(double... values)
{
 double s = 0;
 for (double v : values) { s = s + v; }
 return s;
}
```

can be called as

```
double result = sum(1, -2.5, 3.14);
```

In Java, all parameters are passed by *value*. Each parameter variable is a local variable whose scope extends to the end of the method body. It is initialized with a copy of the value supplied in the call. That value may be a primitive type or a reference type. If it is a reference type, invoking a mutator on the reference will modify the object whose reference has been passed to the method.

Changing the value of the parameter variable has no effect outside the method. Tagging the parameter variable as `final` disallows such a change altogether. This is commonly done to allow access to the parameter variable from an inner class declared in the method.

Java distinguishes between *instance* methods and *static* methods. Instance methods have a special parameter, the *implicit* parameter, supplied in the method call with the syntax

```
implicitParameterValue.methodName(parameterValue1, parameterValue2, . . .)
```

Example:

```
harry.setSalary(30000)
```

The type of the implicit parameter must be the same as the type of the class containing the method definition. A static method does not have an implicit parameter.

In the method body, the `this` variable is initialized with a copy of the implicit parameter value. Using an instance variable name without qualification means to access the instance variable of the implicit parameter. For example,

```
public void setSalary(double s)
{
 salary = s; // i.e., this.salary = s
}
```

By default, Java uses *dynamic method lookup*. The virtual machine determines the class to which the implicit parameter object belongs and invokes the method declared in that class. However, if a method is invoked on the special variable `super`, then the method declared in the superclass is invoked on `this`. For example,

```
public class MyPanel extends JPanel
{
 .
 .
 public void paintComponent(Graphics g)
 {
 super.paintComponent(g);
 // Calls JPanel.paintComponent
 .
 }
 .
}
```

The `return` statement causes a method to exit immediately. If the method type is not `void`, you must return a value. The syntax is

```
return [value];
```

For example,

```
public double getSalary()
{
 return salary;
}
```

A method can call itself. Such a method is called *recursive*:

```
public static int factorial(int n)
{
 if (n <= 1) { return 1; }
 return n * factorial(n - 1);
}
```

# Constructors

A constructor definition has the form

```
modifiers ClassName(parameter1, parameter2, . . .)
 [throws ExceptionType1, ExceptionType2, . . .]
{
 body
}
```

You invoke a constructor to allocate and construct a new object with a new expression

```
new ClassName(parameterValue1, parameterValue2, . . .)
```

A constructor can call the body of another constructor of the same class with the syntax

```
this(parameterValue1, parameterValue2, . . .)
```

For example,

```
public Employee()
{
 this("", 0);
}
```

It can call a constructor of its superclass with the syntax

```
super(parameterValue1, parameterValue2, . . .)
```

The call to this or super must be the first statement in the constructor.

Arrays are constructed with the syntax

```
new ArrayType [= { initializer1, initializer2, . . . }]
```

For example,

```
new int[] = { 1, 4, 9, 16, 25 }
```

When an object is constructed, the following actions take place:

- All instance variables are initialized with 0, false, or null.
- The initializers and initialization blocks are executed in the order in which they are declared.
- The body of the constructor is invoked.

When a class is loaded, the following actions take place:

- All static variables are initialized with 0, false, or null.
- The initializers of static variables and static initialization blocks are executed in the order in which they are declared.

# Statements

A *statement* is one of the following:

- An expression followed by a semicolon
- A branch or loop statement
- A return statement

- A `throw` statement
- A block, that is, a group of variable declarations and statements enclosed in braces  
`{...}`
- A `try` block

Java has two branch statements (`if` and `switch`), three loop statements (`while`, `for`, and `do`), and two mechanisms for nonlinear control flow (`break` and `continue`).

The `if` statement has the form

```
if (condition) statement1 [else statement2]
```

If the `condition` is true, then the first `statement` is executed. Otherwise, the second `statement` is executed.

The `switch` statement has the form

```
switch (expression)
{
 group1
 group2
 .
 .
 [default:
 statement1
 statement2
 .
 .]
}
```

Where each group has the form

```
case constant1:
case constant2:
.
.
statement1
statement2
.
.
```

The `expression` must be an integer, enumeration type, or string. Depending on its value, control is transferred to the first statement following the matching case label, or to the first statement following the `default` label if none of the case labels match. Execution continues with the next statement until a `break` or `return` statement is encountered, an exception is thrown, or the end of the `switch` is reached. Execution skips over any case labels.

The `while` loop has the form

```
while (condition) statement
```

The `statement` is executed while the `condition` is true.

The `for` loop has the form

```
for (initExpression|variableDeclaration;
 condition;
 updateExpression1, updateExpression2, . . .)
 statement
```

The initialization expression or the variable declaration are executed once. While the `condition` remains true, the loop `statement` and the `updateExpressions` are executed.

Examples:

```
for (i = 0; i < 10; i++)
{
 sum = sum + i;
}

for (int i = 0, j = 9; i < 10; i++, j--)
{
 a[j] = b[i];
}
```

The enhanced `for` loop has the form

```
for (Type variable : array|iterableObject)
 statement
```

When this loop traverses an array, it is equivalent to

```
for (int i = 0; i < array.length; i++)
{
 Type variable = array[i];
 statement
}
```

Otherwise, the *iterableObject* must belong to a class that implements the `Iterable` interface. Then the loop is equivalent to

```
Iterator i = iterableObject.iterator();
while (i.hasNext())
{
 Type variable = i.next();
 statement
}
```

The `do` loop has the form

```
do statement while (condition);
```

The *statement* is repeatedly executed until the *condition* is no longer true. In contrast to a `while` loop, the statement of a `do` loop is executed at least once.

The `break` statement exits the innermost enclosing `while`, `do`, `for`, or `switch` statement (not counting `if` or `block` statements).

Any statement (including `if` and `block` statements) can be tagged with a label:

```
label: statement
```

The labeled `break` statement

```
break label;
```

exits the labeled statement.

The `continue` statement skips past the end of the *statement* part of a `while`, `do`, or `for` loop. In the case of the `while` or `do` loop, the loop *condition* is executed next. In the case of the `for` loop, the *updateExpressions* are executed next.

The labeled `continue` statement

```
continue label;
```

skips past the end of the *statement* part of a `while`, `do`, or `for` loop with the matching label.

# Exceptions

The throw statement

```
throw expression;
```

abruptly terminates the current method and resumes control inside the innermost matching catch clause of a surrounding try block. The *expression* must evaluate to a reference to an object of a subclass of `Throwable`.

The try statement has the form

```
try [resourceDeclaration] tryBlock
[catch (ExceptionType1 exceptionVariable1) catchBlock1
catch (ExceptionType2 exceptionVariable2) catchBlock2
. . .]
[finally finallyBlock]
```

- All blocks are block statements in the usual sense, that is, { . . . }-delimited statement sequences.
- An optional resource declaration declares and initializes one or more variables with instances of classes that implement the `AutoCloseable` interface.

The statements in the *tryBlock* are executed. If one of them throws an exception object whose type is a subtype of one of the types in the catch clauses, then its *catchBlock* is executed. As soon as the catch block is entered, that exception is handled.

If the *tryBlock* exits for any reason at all (because all of its statements executed completely; because one of its statements was a `break`, `continue`, or `return` statement; or because an exception was thrown), then the *finallyBlock* is executed.

If the *finallyBlock* was entered because an exception was thrown and it itself throws another exception, then that exception masks the prior exception.

If a resource declaration was present, the `close` method is invoked on all initialized variables when the try block exits normally or because of an exception.

# Packages

A class can be placed in a package by putting the package declaration

```
package packageName;
```

as the first non-import declaration of the source file.

A package name has the form

```
identifier1.identifier2. . .
```

For example,

```
java.util
com.horstmann.bigjava
```

A fully qualified name of a class is

```
packageName.ClassName
```

Classes can always be referenced by their fully qualified class names. However, this can be inconvenient. For that reason, you can reference imported classes by just their *ClassName*. All classes in the package `java.lang` and in the package of the current source file are always imported.

To import additional classes, use an `import` directive

```
import packageName.ClassName;
```

or

```
import packageName.*;
```

The second version imports all classes in the package.

## Generic Types and Methods

A generic type is declared with one or more type parameters, placed after the type name:

```
modifiers class|interface TypeName<typeParameter1, typeParameter2, . . .>
```

Similarly, a generic method is declared with one or more type parameters, placed before the method's return type:

```
modifiers <typeParameter1, typeParameter2, . . .> returnType methodName
```

Each type parameter has the form

```
typeParameterName [extends bound1 & bound2 & . . .]
```

For example,

```
public class BinarySearchTree<T extends Comparable>
public interface Comparator<T>
public <T extends Comparable & Cloneable> T cloneMin(T[] values)
```

Type parameters can be used in the definition of the generic type or method as if they were regular types. They can be replaced with any types that match the bounds. For example, the `BinarySearchTree<String>` type substitutes the `String` type for the type parameter `T`.

Type parameters can also be replaced with *wildcard types*. A wildcard type has the form

```
? [super|extends Type]
```

It denotes a specific type that is unknown at the time that it is declared. For example, `Comparable<? super Rectangle>` is a type `Comparable<S>` for a specific type `S`, which can be `Rectangle` or a supertype such as `RectangularShape` or `Shape`.

## Comments

There are three kinds of comments:

```
/* comment */
// one-line-comment
/** documentationComment */
```

The one-line comment extends to the end of the line. The other comments can span multiple lines and extend to the `*/` delimiter.

Documentation comments are further explained in Appendix F.

# HTML SUMMARY

## A Brief Introduction to HTML

A web page is written in a language called HTML (Hypertext Markup Language). Like Java code, HTML code is made up of text that follows certain strict rules. When a browser reads a web page, the browser *interprets* the code and *renders* the page, displaying characters, fonts, paragraphs, tables, and images.

HTML files are made up of text and *tags* that tell the browser how to render the text. Nowadays, there are dozens of HTML tags—see Table 1 for a summary of the most important tags. Fortunately, you need only a few to get started. Most HTML tags come in pairs consisting of an opening tag and a closing tag, and each pair applies to the text between the two tags. Here is a typical example of a tag pair:

Java is an *<i>object-oriented</i>* programming language.

The tag pair *<i> </i>* directs the browser to display the text inside the tags in *italics*:

Java is an *object-oriented* programming language.

The closing tag is just like the opening tag, but it is prefixed by a slash (/). For example, bold-faced text is delimited by *<b> </b>*, and a paragraph is delimited by *<p> </p>*.

*<p><b>Java</b> is an <i>object-oriented</i> programming language.</p>*

The result is the paragraph

Java is an *object-oriented* programming language.

Another common construct is a bulleted list. For example:

Java is

- object-oriented
- safe
- platform-independent

Here is the HTML code to display it:

```
<p>Java is</p>
object-oriented
safe
platform-independent
```

Each item in the list is delimited by *<li> </li>* (for “list item”), and the whole list is surrounded by *<ul> </ul>* (for “unnumbered list”).

Table 1 Selected HTML Tags			
Tag	Meaning	Children	Commonly Used Attributes
html	HTML document	head, body	
head	Head of an HTML document	title	
title	Title of an HTML document		
body	Body of an HTML document		
h1 ... h6	Heading level 1 ... 6		
p	Paragraph		
ul	Unnumbered list	li	
ol	Ordered list	li	
dl	Definition list	dt, dd	
li	List item		
dt	Term to be defined		
dd	Definition data		
table	Table	tr	
tr	Table row	th, td	
th	Table header cell		
td	Table cell data		
a	Anchor		href, name
img	Image		src, width, height
pre	Preformatted text		
hr	Horizontal rule		
br	Line break		
i or em	Italic		
b or strong	Bold		
tt or code	Typewriter or code font		
s or strike	Strike through		
u	Underline		
super	Superscript		
sub	Subscript		
form	Form		action, method

**Table 1** Selected HTML Tags

Tag	Meaning	Children	Commonly Used Attributes
input	Input field		type, name, value, size, checked
select	Combo box style selector	option	name
option	Option for selection		
textarea	Multiline text area		name, rows, cols

As in Java, you can freely use white space (spaces and line breaks) in HTML code to make it easier to read. For example, you can lay out the code for a list as follows:

```
<p>Java is</p>

object-oriented
safe
platform-independent

```

The browser ignores the white space.

If you omit a tag (such as a `</li>`), most browsers will try to guess the missing tags—sometimes with differing results. It is always best to include all tags.

You can include images in your web pages with the `img` tag. In its simplest form, an image tag has the form

```

```

This code tells the browser to load and display the image that is stored in the file `hamster.jpeg`. This is a slightly different type of tag. Rather than text inside a tag pair `<img> </img>`, the `img` tag uses an attribute to specify a file name. Attributes have names and values. For example, the `src` attribute has the value `"hamster.jpeg"`. Table 2 contains commonly used attributes.

**Table 2** Selected HTML Attributes

Attribute	Description	Commonly Contained in
name	Name of form element or anchor	input, select, textarea, a
href	Hyperlink reference	a
src	Source (as of an image)	img
code	Applet code	applet
width, height	Width, height of image or applet	img, applet
rows, cols	Rows, columns of text area	textarea
type	Type of input field, such as <code>text</code> , <code>password</code> , <code>checkbox</code> , <code>radio</code> , <code>submit</code> , <code>hidden</code>	input
value	Value of input field, or label of submit button	input

**Table 2** Selected HTML Attributes

Attribute	Description	Commonly Contained in
size	Size of text field	input
checked	Check radio button or checkbox	input
action	URL of form action	form
method	GET or POST	form

It is considered polite to use several additional attributes with the `img` tag, namely the *image size* and an *alternate description*:

```

```

These additional attributes help the browser lay out the page and display a temporary description while gathering the data for the image (or if the browser cannot display images, such as a voice browser for blind users). Users with slow network connections really appreciate this extra effort.

Because there is no closing `</img>` tag, we put a slash / before the closing `>`. This is not a requirement of HTML, but it is a requirement of the XHTML standard, the XML-based successor to HTML. (See [www.w3c.org/TR/xhtml1](http://www.w3c.org/TR/xhtml1) for more information on XHTML.)

The most important tag on a web page is the `<a> </a>` tag pair, which makes the enclosed text into a *link* to another file. The links between web pages are what makes the Web into, well, a web. The browser displays a link in a special way (for example, underlined text in blue color). Here is the code for a typical link:

```
Cay Horstmann is the author of this book.
```

When the viewer of the web page clicks on the words [Cay Horstmann](http://horstmann.com), the browser loads the web page located at `horstmann.com`. (The value of the `href` attribute is a *Universal Resource Locator* (URL), which tells the browser where to go. The prefix `http:`, for *Hypertext Transfer Protocol*, tells the browser to fetch the file as a web page. Other protocols allow different actions, such as `ftp:` to download a file, `mailto:` to send e-mail to a user, and `file:` to view a local HTML file.)

**Table 3** Selected HTML Entities

Entity	Description	Appearance
<code>&amp;lt;</code>	Less than	<
<code>&amp;gt;</code>	Greater than	>
<code>&amp;amp;</code>	Ampersand	&
<code>&amp;quot;</code>	Quotation mark	"
<code>&amp;nbsp;</code>	Nonbreaking space	
<code>&amp;copy;</code>	Copyright symbol	©

You have noticed that tags are enclosed in angle brackets (less-than and greater-than signs). What if you want to show an angle bracket on a web page? HTML provides the notations &lt; and &gt; to produce the < and > symbols, respectively. Other codes of this kind produce symbols such as accented letters. The & (ampersand) symbol introduces these codes; to get that symbol itself, use &amp;. See Table 3 for a summary.

You may already have created web pages with a web editor that works like a word processor, giving you a WYSIWYG (what you see is what you get) view of your web page. But the tags are still there, and you can see them when you load the HTML file into a text editor. If you are comfortable using a WYSIWYG web editor, you don't need to memorize HTML tags at all. But many programmers and professional web designers prefer to work directly with the tags at least some of the time, because it gives them more control over their pages.

# GLOSSARY

**Abstract class** A class that cannot be instantiated.

**Abstract data type** A specification of the fundamental operations that characterize a data type, without supplying an implementation.

**Abstract method** A method with a name, parameter variable types, and return type but without an implementation.

**Access specifier** A reserved word that indicates the accessibility of a feature, such as `private` or `public`.

**Accessor method** A method that accesses an object but does not change it.

**Aggregation** The *has-a* relationship between classes.

**Algorithm** An unambiguous, executable, and terminating specification of a way to solve a problem.

**Anonymous class** A class that does not have a name.

**Anonymous object** An object that is not stored in a named variable.

**API (Application Programming Interface)** A code library for building programs.

**API Documentation** Information about each class in the Java library.

**Applet** A graphical Java program that executes inside a web browser or applet viewer.

**Argument** A value supplied in a method call, or one of the values combined by an operator.

**Array** A collection of values of the same type stored in contiguous memory locations, each of which can be accessed by an integer index.

**ArrayList** A Java class that implements a dynamically-growable array of objects.

**Assertion** A claim that a certain condition holds in a particular program location.

**Assignment** Placing a new value into a variable.

**Association** A relationship between classes in which one can navigate from objects of one class to objects of the other class, usually by following object references.

**Asymmetric bounds** Bounds that include the starting index but not the ending index.

**Attribute** A named property that an object is responsible for maintaining.

**Auto-boxing** Automatically converting a primitive type value into a wrapper type object.

**Big-Oh notation** The notation  $g(n) = O(f(n))$ , which denotes that the function  $g$  grows at a rate that is bounded by the growth rate of the function  $f$  with respect to  $n$ . For example,  $10n^2 + 100n - 1000 = O(n^2)$ .

**Binary operator** An operator that takes two arguments, for example `+` in  $x + y$ .

**Binary search** A fast algorithm for finding a value in a sorted array. It narrows the search down to half of the array in every step.

**Bit** Binary digit; the smallest unit of information, having two possible values: 0 and 1. A data element consisting of  $n$  bits has  $2^n$  possible values.

**Black-box testing** Testing a method without knowing its implementation.

**Block** A group of statements bracketed by `{}`.

**Body** All statements of a method or block.

**Boolean operator** An operator that can be applied to Boolean values. Java has three Boolean operators: `&&`, `||`, and `!`.

**Boolean type** A type with two possible values: `true` and `false`.

**Border layout** A layout management scheme in which components are placed into the center or one of the four borders of their container.

**Boundary test case** A test case involving values that are at the outer boundary of the set of legal values. For example, if a method is expected to work for all nonnegative integers, then 0 is a boundary test case.

**Bounds error** Trying to access an array element that is outside the legal range.

**break statement** A statement that terminates a loop or `switch` statement.

**Breakpoint** A point in a program, specified in a debugger, at which the debugger stops executing the program and lets the user inspect the program state.

**Buffer** A temporary storage location for holding values that have been produced (for example, characters typed by the user) and are waiting to be consumed (for example, read a line at a time).

**Bug** A programming error.

## G-2 Glossary

**Byte** A number made up of eight bits. Essentially all currently manufactured computers use a byte as the smallest unit of storage in memory.

**Bytecode** Instructions for the Java virtual machine.

**Call stack** The ordered set of all methods that currently have been called but not yet terminated, starting with the current method and ending with `main`.

**Callback** A mechanism for specifying a block of code so it can be executed at a later time.

**Case sensitive** Distinguishing upper- and lowercase characters.

**Cast** Explicitly converting a value from one type to a different type. For example, the cast from a floating-point number `x` to an integer is expressed in Java by the cast notation `(int) x`.

**catch clause** A part of a `try` block that is executed when a matching exception is thrown by any statement in the `try` block.

**Central processing unit (CPU)** The part of a computer that executes the machine instructions.

**Character** A single letter, digit, or symbol.

**Check box** A user-interface component that can be used for a binary selection.

**Checked exception** An exception that the compiler checks. All checked exceptions must be declared or caught.

**Class** A programmer-defined data type.

**Code coverage** A measure of the amount of source code that has been executed during testing.

**Cohesive** A class is cohesive if its features support a single abstraction.

**Collaborator** A class on which another class depends.

**Collection** A data structure that provides a mechanism for adding, removing, and locating elements.

**Combo box** A user-interface component that combines a text field with a drop-down list of selections.

**Command line** The line the user types to start a program in DOS or UNIX or a command window in Windows. It consists of the program name followed by any necessary arguments.

**Comment** An explanation to help the human reader understand a section of a program; ignored by the compiler.

**Compiler** A program that translates code in a high-level language (such as Java) to machine instructions (such as bytecode for the Java virtual machine).

**Compile-time error** An error that is detected when a program is compiled.

**Component** See **User-interface component**.

**Composition** An aggregation relationship where the aggregated objects do not have an existence independent of the containing object.

**Computer program** A sequence of instructions that is executed by a computer.

**Concatenation** Placing one string after another to form a new string.

**Concrete class** A class that can be instantiated.

**Console program** A Java program that does not have a graphical window. A console program reads input from the keyboard and writes output to the terminal screen.

**Constant** A value that cannot be changed by a program. In Java, constants are defined with the reserved word `final`.

**Construction** Setting a newly allocated object to an initial state.

**Constructor** A sequence of statements for initializing a newly instantiated object.

**Container** A user-interface component that can hold other components and present them together to the user. Also, a data structure, such as a list, that can hold a collection of objects and present them individually to a program.

**Content pane** The part of a Swing frame that holds the user-interface components of the frame.

**Coupling** The degree to which classes are related to each other by dependency.

**CRC card** An index card representing a class that lists its responsibilities and collaborating classes.

**De Morgan's Law** A law about logical operations that describes how to negate expressions formed with *and* and *or* operations.

**Debugger** A program that lets a user run another program one or a few steps at a time, stop execution, and inspect the variables in order to analyze it for bugs.

**Default method** A non-static method that has an implementation in an interface.

**Dependency** The *uses* relationship between classes, in which one class needs services provided by another class.

**Directory** A structure on a disk that can hold files or other directories; also called a folder.

**Documentation comment** A comment in a source file that can be automatically extracted into the program documentation by a program such as javadoc.

**Dot notation** The notation *object.method(arguments)* or *object.variable* used to invoke a method or access a variable.

**Doubly-linked list** A linked list in which each link has a reference to both its predecessor and successor links.

**Dynamic method lookup** Selecting a method to be invoked at run time. In Java, dynamic method lookup considers the class of the implicit parameter *object* to select the appropriate method.

**Editor** A program for writing and modifying text files.

**Encapsulation** The hiding of implementation details.

**Enumeration type** A type with a finite number of values, each of which has its own symbolic name.

**Escape character** A character in text that is not taken literally but has a special meaning when combined with the character or characters that follow it. The \ character is an escape character in Java strings.

**Escape sequence** A sequence of characters that starts with an escape character, such as \n or \".

**Event** See **User-interface event**.

**Event class** A class that contains information about an event, such as its source.

**Event adapter** A class that implements an event listener interface by defining all methods to do nothing.

**Event handler** A method that is executed when an event occurs.

**Event listener** An object that is notified by an event source when an event occurs.

**Event source** An object that can notify other classes of events.

**Exception** A class that signals a condition that prevents the program from continuing normally. When such a condition occurs, an object of the exception class is thrown.

**Exception handler** A sequence of statements that is given control when an exception of a particular type has been thrown and caught.

**Explicit parameter** A parameter of a method other than the object on which the method is invoked.

**Expression** A syntactical construct that is made up of constants, variables, method calls, and the operators combining them.

**Extension** The last part of a file name, which specifies the file type. For example, the extension .java denotes a Java file.

**Fibonacci numbers** The sequence of numbers 1, 1, 2, 3, 5, 8, 13, ..., in which every term is the sum of its two predecessors.

**File** A sequence of bytes that is stored on disk.

**Finally clause** A part of a try block that is executed no matter how the try block is exited.

**Flag** See **Boolean type**.

**Floating-point number** A number that can have a fractional part.

**Flow layout** A layout management scheme in which components are laid out left to right.

**Folder** See **Directory**.

**Font** A set of character shapes in a particular style and size.

**Frame** A window with a border and a title bar.

**Function** A function, in the mathematical sense, yields a result for any assignment of values to its parameters. In Java, functions can be implemented as lambda expressions or instances of functional interfaces.

**Functional interface** An interface with a single abstract method whose purpose is to define a single function.

**Garbage collection** Automatic reclamation of memory occupied by objects that are no longer referenced.

**Generic class** A class with one or more type parameters.

**Graphics context** A class through which a programmer can cause shapes to appear on a window or off-screen bitmap.

**grep** The “global regular expression print” search program, useful for finding all strings matching a pattern in a set of files.

## G-4 Glossary

**Grid layout** A layout management scheme in which components are placed into a two-dimensional grid.

**GUI (Graphical User Interface)** A user interface in which the user supplies inputs through graphical components such as buttons, menus, and text fields.

**Hard disk** A device that stores information on rotating platters with magnetic coating.

**Hardware** The physical equipment for a computer or another device.

**Hash code** A value that is computed by a hash function.

**Hash collision** Two different objects for which a hash function computes identical values.

**Hash function** A function that computes an integer value from an object in such a way that different objects are likely to yield different values.

**Hash table** A data structure in which elements are mapped to array positions according to their hash function values.

**Hashing** Applying a hash function to a set of objects.

**Heapsort algorithm** A sorting algorithm that inserts the values to be sorted into a heap.

**High-level programming language** A programming language that provides an abstract view of a computer and allows programmers to focus on their problem domain.

**HTML (Hypertext Markup Language)** The language in which web pages are described.

**IDE (Integrated Development Environment)** A programming environment that includes an editor, compiler, and debugger.

**Immutable class** A class without a mutator method.

**Implementing an interface** Implementing a class that defines all methods specified in the interface.

**Implicit parameter** The object on which a method is invoked. For example, in the call `x.f(y)`, the object `x` is the implicit parameter of the method `f`.

**Importing a class or package** Indicating the intention of referring to a class, or all classes in a package, by the simple name rather than the qualified name.

**Inheritance** The *is-a* relationship between a more general superclass and a more specialized subclass.

**Initialize** Set a variable to a well-defined value when it is created.

**Inner class** A class that is defined inside another class.

**Input stream** See **Stream (input/output)**.

**Instance method** A method with an implicit parameter; that is, a method that is invoked on an instance of a class.

**Instance of a class** An object whose type is that class.

**Instance variable** A variable defined in a class for which every object of the class has its own value.

**Instantiation of a class** Construction of an object of that class.

**Integer** A number that cannot have a fractional part.

**Integer division** Taking the quotient of two integers and discarding the remainder. In Java the `/` symbol denotes integer division if both arguments are integers. For example, `11/4` is 2, not 2.75.

**Interface type** A type with no instance variables, only abstract or default methods and constants.

**Iterator** An object that can inspect all elements in a container such as a linked list.

**javadoc** The documentation generator in the Java SDK. It extracts documentation comments from Java source files and produces a set of linked HTML files.

**Lambda expression** An expression that defines the parameters and return value of a method in a compact notation.

**Layout manager** A class that arranges user-interface components inside a container.

**Lexicographic ordering** Ordering strings in the same order as in a dictionary, by skipping all matching characters and comparing the first non-matching characters of both strings. For example, “orbit” comes before “orchid” in lexicographic ordering. Note that in Java, unlike a dictionary, the ordering is case sensitive: Z comes before a.

**Library** A set of precompiled classes that can be included in programs.

**Linear search** Searching a container (such as an array or list) for an object by inspecting each element in turn.

**Linked list** A data structure that can hold an arbitrary number of objects, each of which is stored in a link object, which contains a pointer to the next link.

**Literal** A notation for a fixed value in a program, such as `-2`, `3.14`, `6.02214115E23`, `"Harry"`, or `'H'`.

**Local variable** A variable whose scope is a block.

**Logging** Sending messages that trace the progress of a program to a file or window.

**Logical operator** See **Boolean operator**.

**Logic error** An error in a syntactically correct program that causes it to act differently from its specification. (A form of run-time error.)

**Loop** A sequence of instructions that is executed repeatedly.

**Loop and a half** A loop whose termination decision is neither at the beginning nor at the end.

**Machine code** Instructions that can be executed directly by the CPU.

**Magic number** A number that appears in a program without explanation.

**main method** The method that is first called when a Java application executes.

**Map** A data structure that keeps associations between key and value objects.

**Memory location** A value that specifies the location of data in computer memory.

**Merge sort** A sorting algorithm that first sorts two halves of a data structure and then merges the sorted subarrays together.

**Method** A sequence of statements that has a name, may have parameter variables, and may return a value. A method can be invoked any number of times, with different values for its parameter variables.

**Modifier** A reserved word that indicates the accessibility of a feature, such as `private` or `public`.

**Modulus** The `%` operator that computes the remainder of an integer division.

**Mock object** An object that is used during program testing, replacing another object and providing similar behavior. Usually, the mock object is simpler to implement or provides better support for testing.

**Mutator method** A method that changes the state of an object.

**Mutual recursion** Cooperating methods that call each other.

**Name clash** Accidentally using the same name to denote two program features in a way that cannot be resolved by the compiler.

**Nested loop** A loop that is contained in another loop.

**Networks** An interconnected system of computers and other devices.

**new operator** An operator that allocates new objects.

**Newline** The '`\n`' character, which indicates the end of a line.

**No-argument constructor** A constructor that takes no arguments.

**Null reference** A reference that does not refer to any object.

**Number literal** A fixed value in a program this is explicitly written as a number, such as `-2` or `6.02214115E23`.

**Object** A value of a class type.

**Object-oriented programming** Designing a program by discovering objects, their properties, and their relationships.

**Object reference** A value that denotes the location of an object in memory. In Java, a variable whose type is a class contains a reference to an object of that class.

**Off-by-one error** A common programming error in which a value is one larger or smaller than it should be.

**Opening a file** Preparing a file for reading or writing.

**Operating system** The software that launches application programs and provides services (such as a file system) for those programs.

**Operator** A symbol denoting a mathematical or logical operation, such as `+` or `&&`.

**Operator associativity** The rule that governs in which order operators of the same precedence are executed. For example, in Java the `-` operator is left-associative because `a - b - c` is interpreted as `(a - b) - c`, and `=` is right-associative because `a = b = c` is interpreted as `a = (b = c)`.

**Operator precedence** The rule that governs which operator is evaluated first. For example, in Java the `&&` operator has a higher precedence than the `||` operator. Hence `a || b && c` is interpreted as `a || (b && c)`. (See Appendix B.)

**Output stream** See **Stream (input/output)**.

**Overloading** Giving more than one meaning to a method name.

**Overriding** Redefining a method in a subclass.

**Package** A collection of related classes. The `import` statement is used to access one or more classes in a package.

**Panel** A user-interface component with no visual appearance. It can be used to group other components.

## G-6 Glossary

**Parallel arrays** Arrays of the same length, in which corresponding elements are logically related.

**Parameter passing** Specifying expressions to be arguments for a method when it is called.

**Parameter variable** A variable of a method that is initialized with a value when the method is called.

**Partially filled array** An array that is not filled to capacity, together with a companion variable that indicates the number of elements actually stored.

**Path (to a file or directory)** The sequence of directory names and, for a file, a file name at the end, that describes how to reach the file or directory from a given starting point.

**Permutation** A rearrangement of a set of values.

**Polymorphism** Selecting a method among several methods that have the same name on the basis of the actual types of the implicit parameters.

**Primitive type** In Java, a number type or boolean.

**Priority queue** An abstract data type that enables efficient insertion of elements and efficient removal of the smallest element.

**Programming** The act of designing and implementing computer programs.

**Project** A collection of source files and their dependencies.

**Prompt** A string that tells the user to provide input.

**Pseudocode** A high-level description of the actions of a program or algorithm, using a mixture of English and informal programming language syntax.

**Pseudorandom number** A number that appears to be random but is generated by a mathematical formula.

**Public interface** The features (methods, variables, and nested types) of a class that are accessible to all clients.

**Queue** A collection of items with “first in, first out” retrieval.

**Quicksort** A generally fast sorting algorithm that picks an element, called the pivot, partitions the sequence into the elements smaller than the pivot and those larger than the pivot, and then recursively sorts the subsequences.

**Radio button** A user-interface component that can be used for selecting one of several options.

**Reader** In the Java input/output library, a class from which to read characters.

**Recursion** A method for computing a result by decomposing the inputs into simpler values and applying the same method to them.

**Recursive method** A method that can call itself with simpler values. It must handle the simplest values without calling itself.

**Redirection** Linking the input or output of a program to a file instead of the keyboard or display.

**Reference** See **Object reference**.

**Regression testing** Keeping old test cases and testing every revision of a program against them.

**Regular expression** A string that defines a set of matching strings according to their content. Each part of a regular expression can be a specific required character; one of a set of permitted characters such as [abc], which can be a range such as [a-z]; any character not in a set of forbidden characters, such as [^0-9]; a repetition of one or more matches, such as [0-9]+, or zero or more, such as [ACGT]; one of a set of alternatives, such as and|et|und; or various other possibilities. For example, “[A-Za-z][0-9]” matches “Cloud9” or “007” but not “Jack”.

**Relational operator** An operator that compares two values, yielding a Boolean result.

**Reserved word** A word that has a special meaning in a programming language and therefore cannot be used as a name by the programmer.

**Return value** The value returned by a method through a return statement.

**Reverse Polish notation** A style of writing expressions in which the operators are written following the operands, such as 2 3 4 \* + for  $2 + 3 \times 4$ .

**Roundoff error** An error introduced by the fact that the computer can store only a finite number of digits of a floating-point number.

**Run-time error** An error in a syntactically correct program that causes it to act differently from its specification.

**Run-time stack** The data structure that stores the local variables of all called methods as a program runs.

**Scope** The part of a program in which a variable is defined.

**Secondary storage** Storage that persists without electricity, e.g., a hard disk.

**Selection sort** A sorting algorithm in which the smallest element is repeatedly found and removed until no elements remain.

**Sentinel** A value in input that is not to be used as an actual input value but to signal the end of input.

**Sequential search** See **Linear search**.

**Set** An unordered collection that allows efficient addition, location, and removal of elements.

**Shadowing** Hiding a variable by defining another one with the same name.

**Shallow copy** Copying only the reference to an object.

**Shell script** A file that contains commands for running programs and manipulating files. Typing the name of the shell script file on the command line causes those commands to be executed.

**Shell window** A window for interacting with an operating system through textual commands.

**Short-circuit evaluation** Evaluating only a part of an expression if the remainder cannot change the result.

**Side effect** An effect of a method other than returning a value.

**Software** The intangible instructions and data that are necessary for operating a computer or another device.

**Source code** Instructions in a programming language that need to be translated before execution on a computer.

**Source file** A file containing instructions in a programming language such as Java.

**Stack** A data structure with “last-in, first-out” retrieval. Elements can be added and removed only at one position, called the top of the stack.

**Stack trace** A printout of the call stack, listing all currently pending method calls.

**State** The current value of an object, which is determined by the cumulative action of all methods that were invoked on it.

**State diagram** A diagram that depicts state transitions and their causes.

**Statement** A syntactical unit in a program. In Java a statement is either a simple statement, a compound statement, or a block.

**Static method** A method with no implicit parameter.

**Static variable** A variable defined in a class that has only one value for the whole class, and which can be accessed and changed by any method of that class.

**Stream (input/output)** An abstraction for a sequence of bytes from which data can be read or to which data can be written.

**String** A sequence of characters.

**Subclass** A class that inherits variables and methods from a superclass but may also add instance variables, add methods, or redefine methods.

**Substitution principle** The principle that a subclass object can be used in place of any superclass object.

**Superclass** A general class from which a more specialized class (a subclass) inherits.

**Swing** A Java toolkit for implementing graphical user interfaces.

**Symmetric bounds** Bounds that include the starting index and the ending index.

**Syntax** Rules that define how to form instructions in a particular programming language.

**Syntax diagram** A graphical representation of grammar rules.

**Syntax error** An instruction that does not follow the programming language rules and is rejected by the compiler. (A form of compile-time error.)

**Tab character** The '\t' character, which advances the next character on the line to the next one of a set of fixed positions known as tab stops.

**Ternary operator** An operator with three arguments. Java has one ternary operator, `a ? b : c`.

**Test suite** A set of test cases for a program.

**Text field** A user-interface component that allows a user to provide text input.

**Text file** A file in which values are stored in their text representation.

**Throw an exception** Indicate an abnormal condition by terminating the normal control flow of a program and transferring control to a matching catch clause.

**throws clause** Indicates the types of the checked exceptions that a method may throw.

**Token** A sequence of consecutive characters from an input source that belongs together for the purpose of analyzing the input. For example, a token can be a sequence of characters other than white space.

**Trace message** A message that is printed during a program run for debugging purposes.

**try statement** A statement with a body containing one or more statements that are executed until the end of the body is reached or an exception occurs, together with clauses that are invoked when exceptions of a particular type occur.

## G-8 Glossary

**try-with-resources** A version of the try statement whose header initializes a variable of a class that implements the AutoCloseable interface. The close method is invoked on that variable when the try statement terminates, either normally or through an exception.

**Turing machine** A very simple model of computation that is used in theoretical computer science to explore computability of problems.

**Two-dimensional array** A tabular arrangement of elements in which an element is specified by a row and a column index.

**Type** A named set of values and the operations that can be carried out with them.

**Type parameter** A parameter in a generic class or method that can be replaced with an actual type.

**Unary operator** An operator with one argument.

**Unchecked exception** An exception that the compiler doesn't check.

**Unicode** A standard code that assigns code values consisting of two bytes to characters used in scripts around the world. Java stores all characters as their Unicode values.

**Unified Modeling Language (UML)** A notation for specifying, visualizing, constructing, and documenting the artifacts of software systems.

**Uninitialized variable** A variable that has not been set to a particular value. In Java, using an uninitialized local variable is a syntax error.

**Unit testing** Testing a method by itself, isolated from the remainder of the program.

**URL (Uniform Resource Locator)** A pointer to an information resource (such as a web page or an image) on the World Wide Web.

**User-interface component** A building block for a graphical user interface, such as a button or a text field. User-interface components are used to present information to the user and allow the user to enter information to the program.

**User-interface event** A notification to a program that a user action such as a key press, mouse move, or menu selection has occurred.

**Variable** A symbol in a program that identifies a storage location that can hold different values.

**Virtual machine** A program that simulates a CPU that can be implemented efficiently on a variety of actual machines. A given program in Java bytecode can be executed by any Java virtual machine, regardless of which CPU is used to run the virtual machine itself.

**void** A reserved word indicating no type or an unknown type.

**Walkthrough** A step-by-step manual simulation of a computer program.

**White-box testing** Testing methods by taking their implementations into account, in contrast to black-box testing; for example, by selecting boundary test cases and ensuring that all branches of the code are covered by some test case.

**White space** Any sequence of only space, tab, and newline characters.

**Wrapper class** A class that contains a primitive type value, such as Integer.

**Writer** In the Java input/output library, a class to which characters are to be sent.

# INDEX

Page references followed by *t* indicate material in tables.

## Symbols

&& (ampersands), *and* operator, 113–114, 116–117  
\ (backslash), escape character  
  definition, 62  
  in file names, 335  
  in string literals, 62–63  
{ } (braces)  
  enclosing blocks of code, 106  
  lining up, 88  
< > (brackets), diamond syntax, 311  
[ ] (square brackets), specifying  
  array slots, 263  
\$ (dollar sign) in variable names, 35  
. (dot), in dot-separated names,  
  420–421  
= (equal sign), assignment operator,  
  36  
== (equal signs), relational operator  
  comparing strings, 93–94  
  equality testing, 90–91  
  *versus equals* method, 92  
  syntax, 91*t*  
  testing for *null* references, 405  
! (exclamation point), *not* operator,  
  114  
!= (exclamation point, equal), *not*  
  *equal* relational operator, 91*t*  
< (less than), relational operator, 91*t*  
<= (less than or equal), relational  
  operator, 91*t*  
> (greater than), relational operator,  
  91*t*  
>= (greater than or equal), relational  
  operator, 91*t*  
( ) (parentheses)  
  unbalanced, 48  
% (percent sign)  
  in variable names, 35  
  modulus operator, 44  
  negative remainders, 49  
+ (plus sign), concatenation operator,  
  61  
? (question mark) in variable names,  
  35

" (quotation marks), string delimiters  
  definition, 12  
  in string literals, 62  
; (semicolon)  
  ending statements, 12  
  omitting, 13  
// (slash slash), comment indicator,  
  38  
/\*\*...\*/ (slash, asterisks...), comment  
  delimiters, 217  
\_ (underscore), in variable names, 35  
|| (vertical bars), *or* operator,  
  113–114, 116–117

## A

abs method, `java.lang.Math` class, 46*t*,  
  A-17  
absolute values, calculating. *See abs*  
  method, `java.lang.Math` class.  
abstract classes, 456–457

## I-2 Index

- approximate solutions, finding, 184–185
- arguments
  - definition, 12, 213
  - of method call, 213
  - modifying, 219–220
- Ariane rocket incident, 361
- arithmetic operations. *See also* data types; *specific operations*.
  - (minus signs), decrement operator, 43
  - / (slash), division symbol, 43
  - \* (asterisk), multiplication symbol, 43
  - % (percent sign), modulus operator, 44
  - ++ (plus signs), increment operator, 43
  - combining with assignment, 49
  - expressions, 43
  - integer division, 44, 49
  - mathematical methods, 45–46
  - modulus, 44
  - powers, 45–46
  - precedence, 43
  - roots, 45–46
  - symbols for, 43
  - unintended integer division, 48
- ARPANET, 481
- array elements
  - duplicates, removing (Video Example), 294
  - iterating through, 269–270
- array lists. *See also* `java.util.ArrayList<E>` class.
  - adding/removing elements, 303–304
  - versus* arrays, 308–309t
  - auto-boxing, 305–306
  - collecting values, 412
  - constructors, 302
  - converting array algorithms, 307
  - copying, 305
  - current size, getting, 303
  - declaring, 302–304
  - definition, 301
  - diamond syntax, 311
  - enhanced for loop, 304
  - full code example, 308
  - generic class, 302
  - inserting elements, 307
  - iterating through, 304
  - length, determining, 311
  - as method arguments, 305
  - as method return values, 305
  - overview, 302–304
  - removing matches, 307–308
  - size, determining, 311
  - syntax, 302
  - type parameter, 302
  - wrapper classes, 305–306
- `ArrayList` constructor, `java.util.ArrayList<E>` class, A-20
- arrays. *See also* enumeration types.
  - [ ] (square brackets), specifying array slots, 263
  - versus* array lists, 308–309t
  - bounds errors, 264, 267
  - collecting values, 412
  - companion variables, 266
  - current size, 266
  - declaring, 262–265
  - definition, 262
  - fixed length, 265
  - full code example, 266
  - How To example, 287–290
  - indexes, 263
  - initialization, 262–264, 267
  - length, determining, 266, 311
  - with methods, 280–283
  - multidimensional, 301
  - overview, 262–265
  - partially filled, 266
  - printing, 298–299
  - references, 265
  - sequences of related values, 268
  - size, determining, 311
  - sorting, 279
  - syntax, 263
  - underestimating data size, 279
  - unfilled, 267
  - uninitialized, 267
  - variable row lengths, 300–301
  - world population table (Worked Example), 300
- arrays, common algorithms
  - averaging elements, 271
  - binary search, 272, 279–280
  - combining and adapting, 279–284
  - converting array algorithms to array lists, 307
  - copying arrays, 275–276
  - developing, 284–287
  - discovering by manipulating physical objects, 291–294
  - element position, finding, 272
  - element separators, 271–272
  - filling, 270
- full code examples, 281, 287, 293
- growing arrays, 275–276
- inserting elements, 273
- linear search, 272
- maximum value, finding, 271
- minimum value, finding, 271
- reading input, 276–278
- removing elements, 272–273
- resizing arrays, 275–276
- reusing, 279–284
- rolling dice (Worked Example), 290
- searching for values, 272
- sequential search, 272
- summing elements, 271
- swapping elements, 274
- switching array halves, 293
- arrays, elements
  - counting, 265, 266. *See also* companion variables.
  - definition, 263
  - numbering, 264
  - position, finding, 272
- arrays, two-dimensional
  - accessing elements, 295–296
  - computing row and column totals, 297
  - declaring, 295
  - definition, 295
  - locating neighboring elements, 296
  - parameters, 298–300
  - syntax, 295
- artificial intelligence, 121
- `asin` method, `java.lang.Math` class, A-18
- assertions, 360
- assignment statements
  - = (equal sign), assignment operator, 36
  - combining with arithmetic operations, 49
  - definition, 36
  - example, 38–39
  - syntax, 36
- associativity, Java operators, A-5
- asymmetric bounds, 159, 160
- `atan` method, `java.lang.Math` class, A-18
- `atan2` method, `java.lang.Math` class, A-17
- auto-boxing, 305–306

averages  
 array elements, 271  
 computing with loops, 167

## B

baby name analysis (Worked Example), 350  
 backing up files, 10  
 backslash (\) escape character, in string literals, 62–63  
 BankAccount class, full code example, 399  
 BankAccountTester class, full code example, 399  
 bar charts, drawing  
   description, 520–524  
   Worked Example, 528  
 Berners-Lee, Tim, 481  
 big numbers, 42  
 binary data, reading and writing, 336–337  
 binary search, 272, 279–280  
 binarySearch method  
   java.util.Arrays class, A-20–21  
   java.util.Collections class, A-21  
 black box, methods, 212–213  
 body, methods, 214  
 Boolean constructor, java.lang.  
   Boolean class, A-15  
 Boolean operators  
   ! (exclamation point), *not* operator, 114  
   && (ampersands), *and* operator, 113–114, 116–117  
   || (vertical bars), *or* operator, 113–114, 116–117  
 De Morgan’s Law, 117–118  
 flags, 113  
 full code example, 114  
 inverting conditions, 114  
 precedence, 113  
 short circuiting evaluation of, 116–117  
 Boolean variables, in loops, 161  
 BorderLayout constructor, java.awt.  
   BorderLayout class, A-11  
 boundary test cases, 110–111  
 bounds errors, arrays, 264, 267  
 bounds for loops, choosing, 158  
 boxString method (full code example), 224

braces ({ })  
   enclosing blocks of code, 106  
   lining up, 88  
 brackets (< >), diamond syntax, 311  
 break statements, 162–163  
 buffer overrun attacks, 268  
 bugs, historical first, 148. *See also* debugging.  
 ButtonFrame1.java class, 501  
 ButtonFrame2.java class, 502–503  
 ButtonViewer1.java class, 501  
 bytes, reading/writing one at a time, 336–337

## C

Caesar, Julius, 345  
 CaesarCipher.java class, 346–347  
 calculating a course grade (Worked Example), 236  
 calling  
   constructors, 392, 408  
   methods. *See* methods, calling.  
   superclass methods, 447  
 cars, autonomous vehicles, 121  
 case sensitivity  
   definition, 9  
   errors caused by, 15  
   misspelling words, 15  
 cash register simulation, 381–383  
 CashRegister class (full code examples), 382, 386, 388  
 CashRegister.java class  
   code, 391  
   full code example, 401  
 CashRegisterTester.java class, 394–395  
 cast operator  
   converting floating-point numbers to integer, 46–47  
   syntax, 46  
 catch clause, 354–355, 359  
 catching exceptions, 354–355  
 “The Cathedral and the Bazaar,” 402  
 ceil method, java.lang.Math class, A-17  
 cell phone plan, evaluating (Video Example), 163  
 central processing unit (CPU), 3  
 character patterns. *See* regular expressions.  
 characters. *See also* strings.  
   classifying, 338

definition, 63  
 international alphabets, 68  
 primitive types, 65  
 reading, 338  
 reading from a string, 340  
*versus* strings, 63  
 Unicode, 68

ChartComponent2.java class, 519  
 ChartComponent.java class, 516, 522  
 ChartViewer2.java class, 520  
 ChartViewer.java class, 516–517  
 checked exceptions, 356  
 ChoiceQuestion.java class, 448–449  
 class files, 9  
 class names, naming conventions, 35  
 classes. *See also* OOP (object-oriented programming); subclasses; superclasses.  
   abstract, 456–457  
   concrete, 457  
   definition, 11, 376–377  
   extending. *See* inheritance.  
   final, 457  
   importing from packages, 51  
   *versus* interface types, 471  
   modifiers, definition, 379  
   naming conventions, 35  
   organizing into packages, 417  
   private implementation, 382  
   public. *See* classes, public  
     interfaces.  
   tester, 394  
   testing, 393–399  
 classes, implementing  
   cash register, 381–383  
 CashRegister class (full code examples), 382, 386, 388  
 Counter class (full code example), 380  
 description, 378–381  
 full code example, 399  
 How To, 395–399  
 a menu class (Worked Example), 399  
 tally counter, 378–381  
 classes, instance variables  
   declaration syntax, 379  
   definition, 379  
   description, 379–381  
   designing data representation, 385–386  
   modifiers, 379  
   type, 379

## I-4 Index

- classes, public interface
  - accessor methods, 383
  - components of, 382
  - definition, 377
  - mutator methods, 383
  - specifying, 381–384
- `ClickListener.java` class, 499–500
- `clone` method, `java.lang.Object` class, A-18
- `close` method
  - `java.io.InputStream` class, A-14
  - `java.io.OutputStream` class, A-15
  - `java.io.PrintStream` class, A-15
  - `java.io.PrintWriter` class, 333–334, A-15
  - `java.util.Scanner` class, A-24
- code coverage, 110
- code replication, 225–227
- collecting values
  - with arrays or array lists, 412
  - patterns for object data, 412
- color, drawing, 517–520
- `Color` constructor, `java.awt.Color` class, A-11
- command line arguments, 345–347
- comments
  - `//` (slashes), delimiter, 37–38
  - `/*... */` (slash asterisk...)
  - delimiters, 38
  - `/**... */` (slash asterisk asterisk...)
  - delimiters, 38
- converting to documentation. *See javadoc utility.*
- definition, 37–38
- `long`, 38
- method, 217
- `short`, 38
- companion variables, 266
- Comparable interface type, 474–475
- `compare` method
  - `java.lang.Double` class, A-16
  - `java.lang.Integer` class, A-16
  - `java.util.Comparator<T>` interface, A-21
- `compareTo` method
  - comparing strings, 94
  - example, 474–475
  - `java.lang.Comparable<T>` interface, A-16
  - `java.lang.String` class, A-18
- `comparing` method, `java.util.Comparator<T>` interface, A-22
- comparisons
  - adjacent values, with loops, 170–171
  - floating-point numbers, 93–94
  - full code example, 92
  - integers and floating-point numbers, 475–476
  - lexicographic ordering, 94–95
  - object contents. *See compareTo method; equals method.*
  - relational operators, 90–91, 115–116
  - strings, 92, 94–95
  - syntax, 91
- compilation process, 9
- compilers, 6
- compile-time errors, 14
- compiling and running programs, Video Example, 11
- computations
  - cost of stamps, Worked Example, 58
  - distance on Earth, Video Example, 67
  - How To, 56–58
  - travel time, Worked Example, 60
- computer programs, definition, 2. *See also Java programs; programming; software.*
- computer viruses, 268
- computers
  - anatomy of, 3–5
  - common uses for, 5
  - description, 2
  - history of. *See history of computers.*
  - PCs (personal computers), schematic design of, 4
- concatenating strings, 61
- concrete classes, 457
- conditional operators, 89
- constants
  - declaring, 37
  - example, 38–39
  - in interface types, 476
  - versus* magic numbers, 41
  - naming conventions, 37
  - syntax, 37
- constructors
  - automatic generation, 390
  - calling, 392
  - calling one from another, 408
  - declaring as void, 393
- definition, 389
- naming, 389
- new reserved word, 392
- returning values, 389
- subclasses, 451
- superclasses, 451–452
- syntax, 390
- void reserved word, 393
- constructors, initializing instance variables
  - default values, 390
  - overview, 389
  - syntax, 390
- `contains` method, `java.util.Collection<E>` interface, A-21
- converting data types. *See also cast operator.*
  - array list algorithms from array algorithms, 307
  - full code example, 47
  - numbers from strings, 340. *See also parseInt method.*
  - strings from objects. *See toString method.*
  - strings to numbers, 340. *See also parseInt method.*
- copying
  - array lists, 305
  - arrays, 275–276
  - files. *See backing up files.*
  - object references, 404
- `copyOf` method, `java.util.Arrays` class, 275–276, A-21
- `cos` method, `java.lang.Math` class, 45*t*, A-17
- cosine of  $x$ , calculating. *See cos method, java.lang.Math class.*
- count-controlled loops, 153
- Counter class (full code example), 380
- counters
  - in loops, 167–168
  - tally counter, 378–381
- counting
  - elements of arrays, 265, 266
  - loop iterations, 158
- counting events, patterns for object data, 411–412
- course grade, calculating (Worked Example), 236
- CPU (central processing unit)
  - definition, 3
  - simulating. *See virtual machine.*

credit card processing (Worked Example), 174  
 cube volume, computing, 214–216  
`Cubes.java` class, 216  
`currentTimeMillis` method, `java.lang.System` class, A-19

## D

dangling `else` problem, 106  
`DARPA`, 481  
`DARPA Urban Challenge`, 121  
 data types. *See also* arithmetic operations.  
     character type, 63  
     number types, 34–35, 41–42  
     numbers with fractions. *See*  
         double data type; floating-point numbers.  
     numbers without fractions. *See*  
         int data type.  
     primitive, 41–42, 66–67, 302,  
         305–306, 403  
     string type, 61  
     testing for, 466–468. *See also*  
         `instanceof` operator.  
`DataAnalyzer.java` class, 361–365  
 database access, SQL package, 417t  
`Date` constructor, `java.util.Date` class, A-22  
 De Morgan, Augustus, 117  
 De Morgan’s Law, 117–118  
 debugging  
     debuggers, 239–240  
     the first bug, 148  
     test cases, 110–111  
     tester classes, 394  
     unit tests, 393  
 decimal logs, calculating. *See* `log10` method, `java.lang.Math` class.  
 decisions. *See also* comparisons; if statements.  
     Boolean operations, 113–118  
     conditional operators, 89  
 declaring  
     array lists, 302–304  
     arrays, 262–265  
     classes, 11  
     instance variables, 379  
     interfaces, 470–472  
     variables, 32–34t  
 decrementing variables, 43  
 default package, 418  
 definite loops, 153

degrees, converting to radians. *See*  
     `toRadians` method, `java.lang.Math` class.  
 Denver’s luggage handling system, 97  
 dialog boxes  
     file choosing, 335–336  
     full code example, 67, 336  
     for input/output, 67  
 diamond syntax, 311  
 dice. *See* rolling dice.  
`Dice.java` class, 183–184  
 die throws, simulating, 183–184  
`digitSum` method, full code example, 244  
`Dimension` constructor, `java.awt.Dimension` class, A-12  
 directories, 9–10. *See also* files.  
 discussion board, Video Example, 463  
`divide` method  
     `java.math.BigDecimal` class, A-20  
     `java.math.BigInteger` class, A-20  
 dividing household expenses, algorithm for, Video Example, 22  
 division  
     in Java, 43, 47t  
     integer division, 44  
     unintended integer division, 48  
     Video Example, 49  
 “Division by zero” errors, 14  
 do loops, 158–159  
 Document Object Model package, 417t  
 documentation  
     for APIs. *See* API documentation.  
     converting comments to. *See*  
         javadoc utility.  
 dollar sign (\$) in variable names, 35  
 dongles, 188  
 dot notation in methods, 66  
 double data type  
     converting to int, 46–47. *See also*  
         cast operator.  
     definition, 34t  
     wrapping to Double, 306  
`Double` constructor, `java.lang.Double` class, A-16  
`DoubleInvestment.java` class, 145  
`doubleValue` method, `java.lang.Double` class, A-16  
 drawing  
     bar charts, description, 520–524  
     bar charts, Worked Example, 528  
     color, 517–520  
     `drawLine` method, 517, 518  
     `drawOval` method, 517  
     `drawRect` method, 517  
     `drawString` method, 518  
     `fillOval` method, 517  
     flags, 525–528  
     full code example, 527  
     geometric shapes, Video Example, 480  
     graphical shapes, 186–188  
     `Graphics` class, 516–517  
     How To, 525–528  
     lines, 517–520  
     methods for, 186–188  
     ovals, 517–520  
     `paintComponent` method, 521–524  
     painted components, 521–524  
     rectangles, 515–517  
     repaint method, 521–524  
     text, 517–520  
     on user-interface components, 515–520  
     visualizing investment growth, example, 520–524  
`drawLine` method, `java.awt.Graphics` class, 517, 518, A-12  
`drawOval` method, `java.awt.Graphics` class, 517, A-12  
`drawRect` method, `java.awt.Graphics` class, 517, A-12  
`drawString` method, `java.awt.Graphics` class, 518  
 duplicate array elements, removing (Video Example), 294  
 duplicate code  
     in branches, 90  
     eliminating, 225–227  
 dynamic method lookup, 453

## E

earthquake descriptions  
     full code example, 100  
     Loma Prieta quake, 98  
     Richter scale, 98t  
 editors, definition, 8  
 electronic numerical integrator and computer (ENIAC), 5  
 electronic voting machines, 416

## I-6 Index

- elements of arrays
    - counting, 265, 266
    - definition, 263
    - numbering, 264
  - ElevatorSimulation2.java class, 119–120
  - ElevatorSimulation.java class, 86
  - else statements, dangling else problem, 106
  - empty strings
    - definition, 61
    - versus null* references, 405
  - EmptyFrameViewer.java class, 495
  - encapsulation, definition, 377
  - encryption algorithms, 351
  - English words
    - computing plurals of (Video Example), 107
    - for numbers, 230–234
  - enhanced for loops. *See also* for loops.
    - array lists, 304
    - iterating through arrays, 269–270
  - ENIAC (electronic numerical integrator and computer), 5
  - enumeration types, 107. *See also* arrays.
  - EOFException constructor, java.io.EOFException, A-14
  - equal sign (=), assignment operator, 36
  - equal signs (==), relational operator
    - comparing strings, 92, 94–95
    - equality testing, 90–91
    - versus equals* method, 92
    - syntax, 91*t*
    - testing for *null* references, 405
  - equality testing. *See* equal signs (==); equals method.
  - equals method
    - comparing objects, 465–466
    - inheritance, 469
    - java.lang.Object class, A-18
    - java.lang.String class, 92, A-18
  - equalsIgnoreCase method, java.lang.String class, A-18
  - error handling, input errors, 361–366. *See also* exception handling.
  - error messages
    - logging, 112
    - trace messages, 112
  - errors
    - arrays, 267
    - compile-time, 14
    - dangling else problem, 106
    - “Division by zero,” 14
    - exceptions, 14
    - full code example, 14
    - logic, 14
    - in loops, 146–148
    - misspelling words, 15
    - roundoff, 93–94
    - run-time, 14
    - syntax, 14
    - unintended integer division, 48
  - ESA (European Space Agency), 361
  - escape sequence
    - definition, 335
    - in file names, 335
    - in string literals, 62–63
  - evaluating a cell phone plan (Video Example), 163
  - event handling, lambda expressions for, 509
  - event listeners
    - description, 499–501
    - forgetting to attach, 506
    - frames as, 506–507
    - inner classes for, 501–503
  - event sources, 499–501
  - event-controlled loops, 153
  - events
    - action listeners, 500–501
    - tracking an investment, example, 503–504
  - exception handlers, 353
  - exception handling. *See also* error handling.
    - application input errors, 361–366
    - catch clause, 354–355, 359
    - catching exceptions, 354–355
    - checked exceptions, 356
    - closing resources, 357–358
    - definition, 352
    - finally statement, 360–361
    - internal errors, 355
    - reading exception reports, 286–287
    - sample application, 361–366
    - squelching exceptions, 359
    - throw early, catch late, 359
    - throwing exceptions, 352–353, 359
  - throws clause, 356–357
  - try blocks, 354–355
  - try/catch statements, 354–355
  - try/finally statement, 360–361
  - try-with-resources statement, 357–358
  - unchecked exceptions, 355
  - exception handling, full code examples
    - catching exceptions, 355
    - checked exceptions, 356
    - closing resources, 358
    - throwing exceptions, 352
  - exceptions, definition, 14
  - exclamation point, equal (!=), *not equal* relational operator, 91*t*
  - exclamation point (!), *not* operator, 114
  - executable algorithms, 16
  - exists method, java.io.File class, A-14
  - exit method, java.lang.System class, A-19
  - exp method, java.lang.Math class, 45*t*, A-17
  - expressions, spaces in, 49
  - extending classes. *See* inheritance.
  - extends reserved word, 443
- ## F
- file choosers, 335–337
  - File class, 332
  - file dialog boxes
    - choosing files from a list, 335–336
    - full code example, 336
  - File constructor, java.io.File class, A-14
  - file names, as string literals, 335
  - FileInputStream constructor, java.io.FileInputStream class, A-14
  - FileNotFoundException, 333–334, 356, 362–365
  - FileOutputStream constructor, java.io.FileOutputStream class, A-14
  - files. *See also* folders.
    - backing up, 10
    - definition, 9
  - FilledFrameViewer.java class, 495–496
  - filling arrays, 270
  - fillRect method, java.awt.Graphics class, 517, A-12
  - fillRect method, java.awt.Graphics class, A-12

final classes, 457  
 final methods, 457  
 final reserved word, 457  
 finally statement, 360–361  
 first kit computer, 228  
 fixed length arrays, 265  
 flags  
   Boolean, 113  
   drawing, 525–528  
 floating-point numbers. *See also*  
   double data type.  
   comparing, 93–94  
   comparing to integers, 475–476  
   converting to integers, 46–47. *See also*  
     cast operator.  
   definition, 34  
   Pentium bug, 50  
   roundoff errors, 40–41  
   wrapper class for, 306t  
**floor** method, `java.lang.Math` class, A-17  
**floorMod** method, `java.lang.Math` class, A-17  
 flowcharting, loops, 159  
 flowcharts, 107–110. *See also*  
   storyboards.  
**FlowLayout** constructor, `java.awt.FlowLayout` class, A-12  
 folders, 9–10. *See also* files.  
**Font** constructor, `java.awt.Font` class, A-12  
**for** loops, 154–158. *See also* enhanced  
   for loops; while loops.  
**for** statement, syntax, 154  
 formal parameters, 217–220  
 format flags, 342t  
**format** method, `java.lang.String` class, A-19  
 format specifiers, 52–55, 342–343  
 format types, 343t  
 formatting output  
   format flags, 342t  
   format specifiers, 52–55, 342–343  
   format types, 343t  
   full code example, 343  
   overview, 342–343  
**forName** method, `java.lang.Class` class, A-16  
 fractional numbers. *See* floating-point numbers.  
 frame class, adding `main` method to, 498

frame windows  
   adding user-interface  
     components, 495–496  
     as event listeners, 506–507  
     combining classes, 498  
     constructing, 494–495  
     definition, 494  
     displaying, 494–495  
     for complex frames, 497–498  
     frame size, setting, 497  
     full code example, 497  
     grouping user-interface  
       components, 496  
       `main` method, 498  
       panels, 496  
 free software, 402  
 function expressions. *See* lambda expressions.  
 function objects, 478–479  
 functional interfaces, 479–480

**G**

**Gallery6.java** class, 181  
 generic class, array lists, 302  
 generic interface types, 476–477  
 genetic code, decoding (Video Example), 120  
 geometric shapes, drawing (Video Example), 480  
**get** method  
   `java.util.ArrayList<E>` class, 303, A-20  
   `java.util.Calendar` class, A-21  
   `java.util.Map<K, V>` interface, A-23  
**getFirst** method, `java.util.LinkedList<E>` class, A-22  
**getGlobal** method, `java.util.logging.Logger` class, 112, A-24  
**getHeight** method  
   `java.awt.Component` class, A-11  
   `java.awt.Rectangle` class, A-13  
**getStrokeForEvent** method, `javax.swing.KeyStroke` class, A-27  
**getLast** method, `java.util.LinkedList<E>` class, A-22  
**getMessage** method, `java.lang.Throwable` class, A-19  
**getSelectedFile** method, `javax.swing.JFileChooser` class, 336, A-25  
**getSelectedItem** method, `javax.swing.JComboBox` class, A-25

getSource method, `java.util.EventObject` class, A-22  
**getText** method, `javax.swing.text.JTextComponent` class, A-27  
**getValue** method, `javax.swing.JSlider` class, A-26  
**getWidth** method  
   `java.awt.Component` class, A-11  
   `java.awt.Rectangle` class, A-13  
**getX** method  
   `java.awt.event.MouseEvent` class, A-14  
   `java.awt.Point` class, A-13  
   `java.awt.Rectangle` class, A-13  
**getY** method  
   `java.awt.event.MouseEvent` class, A-14  
   `java.awt.Point` class, A-13  
   `java.awt.Rectangle` class, A-13  
 GNU GPL (GNU General Public License), 402  
 GNU project, 402  
 Gosling, James, 6  
**Graphics** class, 516–517  
 greater than or equal ( $\geq$ ), relational operator, 91t  
 greater than ( $>$ ), relational operator, 91t  
**GregorianCalendar** constructor, `java.util.GregorianCalendar` class, A-22  
 grep program, 344  
**GridLayout** constructor, `java.awt.GridLayout` class, A-13  
**grow** method, `java.awt.Rectangle` class, A-13  
 GUI (graphical user interface)  
   description, 498  
   events, 499  
 Gutenberg project, 481

**H**

hand-tracing  
   description, 105–106  
   loops, 149–152  
   methods, 234–235  
   objects, 399–402  
 handling input errors, 361  
 hard disk, illustration, 3–4  
 hardware  
   definition, 2  
   evolution of, 5

## I-8 Index

- HashMap constructor, `java.util.`  
  `HashMap<K, V>` class, A-22
- HashSet constructor, `java.util.`  
  `HashSet<E>` class, A-22
- hasNext method  
  `java.util.Iterator<E>` interface, A-22  
  `java.util.Scanner` class, A-24
- hasNextDouble method, `java.util.`  
  `Scanner` class, 118, A-24
- hasNextInt method, `java.util.Scanner`  
  class, 118–119, A-24
- hasNextLine method, `java.util.`  
  `Scanner` class, A-24
- hasPrevious method, `java.util.`  
  `ListIterator<E>` interface, A-23
- headers, methods, 214
- “Hello, World” program  
  analyzing, 11–13  
  source code, 11  
  writing, 8–10
- `HelloPrinter.java` class, 11
- high-level languages, 6
- history of computers  
  Altair 8800, 228  
  Apple II, 228  
  first kit computer, 228  
  hardware evolution, 5  
  microprocessors, 228  
  personal computing, 228  
  VisiCalc, 228
- Hoff, Marcian E., 228
- I**
- if statements. *See also* switch statements.  
  dangling else problem, 106  
  definition, 84  
  duplicate code in branches, 90  
  ending with a semicolon, 88  
  flowchart for, 85  
  implementing (How To), 95–97  
  input validation, 118–120  
  multiple alternatives, 98–101  
  nesting, 102–107  
  sample program, 85  
  syntax, 86
- `IllegalArgumentException`, `java.lang.`  
  `IllegalArgumentException` class, 352–353, A-16
- `ImageIcon` constructor, `javax.swing.`  
  `ImageIcon` class, A-25
- implementation, classes. *See* classes, implementing.
- implements reserved word, 472–473
- implicit parameters, 455–456
- import statement, 418–419
- importing  
  classes from packages, 51  
  packages, 51, 418–419
- in object, `java.lang.System` class, A-19
- income tax computation, 102–104
- income tax rate schedule, 102t
- incrementing/decrementing variables, 43
- indefinite loops, 153
- indenting code, with tabs, 89
- indexes, arrays, 263
- `IndexOutOfBoundsException`, 355–356
- infinite loops, 147
- INFO message level, 112
- info method, `java.util.logging.Logger` class, 112, A-24
- information hiding. *See* encapsulation.
- inheritance. *See also* polymorphism.  
  definition, 438  
  drawing geometric shapes, Video Example, 480  
  equals method, 469  
  full code example, 445, 463  
  indicating, 443  
  purpose of, 442  
  super reserved word, 447, 451  
  Video Example, 463
- inheritance hierarchy  
  developing (How To), 458–463  
  for elements used in this book, A-9–11  
  Worked Example, 458–463
- initializing  
  arrays, 262–264, 267  
  variables, 39–40
- initializing, instance variables. *See also* constructors.  
  default values, 390  
  overview, 389  
  syntax, 390
- `Initials.java` class, 64–65
- inner classes  
  anonymous, 508–509  
  for event listeners, 501–503  
  local, 507–508
- input. *See also* `java.util.Scanner` class; output.  
  definition, 4  
  dialog boxes, 67  
  from a keyboard, 50–52, 62  
  prompting for, 51  
  reading, 50–52  
  reading strings from the console, 62  
  redirecting, 163
- input redirection, 163
- input statements, syntax, 51
- input validation, 118–120
- input/output, package for, 417
- `InputStreamReader` constructor, `java.io.InputStreamReader` class, A-15
- instance methods  
  implementing, 386–388  
  overview, 66–67  
  syntax, 387
- instance variables  
  declaration syntax, 379  
  definition, 379  
  description, 379–381  
  designing data representation, 385–386  
  modifiers, 379  
  package access, 421  
  public *versus* private, 388  
  type, 379
- instanceof operator  
  description, 466–468  
  full code example, 467
- int data type  
  converting from double, 46–47  
  definition, 34  
  overflow, 40  
  range of, 40  
  wrap to Integer, 306
- integer division, 49
- `IntegerName.java` class, 232–233
- integers  
  comparing to floating-point numbers, 475–476  
  converting from floating-point numbers, 46–47. *See also* cast operator.  
  unintended integer division, 48
- integrated development environment, 8
- Intel Corporation, 50

interface types  
*versus* classes, 471  
**Comparable**, 474–475  
**compareTo** method, 474–475  
defining an interface, 470–472  
definition, 471  
generic, 476–477  
implementing interfaces, 472–473  
**implements** reserved word, 472–473  
specifying constants in, 476  
syntax, 471  
interfaces  
default methods, 477–478  
defining, 470–472  
full code examples, 477, 478, 479  
function objects, 478–479  
functional, 479–480  
implementing, 472–473, 506  
investigating number sequences,  
  Worked Example, 480  
lambda expressions, 479–480  
modifying parameter types in the  
  implementing method, 506  
Sequence, Worked Example, 480  
static methods, 477  
international alphabets, 68  
Internet, history of, 470  
Internet Explorer, 470  
intersection method, `java.awt.Rectangle` class, A-13  
**intValue** method, `java.lang.Integer` class, A-16  
investigating number sequences,  
  Worked Example, 480  
`InvestmentFrame.java`, 504–505  
`InvestmentFrame2.java` class, 510–511  
`InvestmentFrame3.java` class, 513–514  
`InvestmentFrame4.java` class, 522–524  
`InvestmentTable.java` class, 155–156  
`InvestmentViewer.java`, 505  
“is-a” relationship. *See* inheritance.  
**isDigit** method, `java.lang.Character` class, A-15  
**isEditable** method  
  `javax.swing.JComboBox` class, A-25  
  `javax.swing.text.JTextComponent` class, A-27  
**isLetter** method, `java.lang.Character` class, A-15  
**isLowerCase** method, `java.lang.Character` class, A-15

**isSelected** method, `javax.swing.AbstractButton` class, A-24  
**isUpperCase** method, `java.lang.Character` class, A-16  
**iterator** method, `java.util.Collection<E>` interface, A-21

**J**

Java language. *See also* applets; programming;  
description, 6–7  
high-level languages, 6  
integrated development environment, 8  
portability, 7  
versions, 6t

Java library, 6–7. *See also* packages.

Java operators, summary of, A-5t–6t

Java programs  
class files, 9  
compilation process, 9  
compilers, 6  
compiling and running (Video Example), 11  
getting started. *See* “Hello, World” program.  
machine code, 6  
source code, 9  
source files, 9  
syntax, 12

Java virtual machine, definition, 7

`java.applet` package, 417

`java.awt` package, 417, A-11–13

`java.awt.BorderLayout` class, method summary, A-11

`java.awt.Color` class, method summary, A-11

`java.awt.Component` class, method summary, A-11

`java.awt.Container` class, method summary, A-12

`java.awt.Dimension` class, method summary, A-12

`java.awt.event` package, A-13–14

`java.awt.event.ActionListener` interface, method summary, A-13

`java.awt.event.KeyEvent` class, description, A-13

`java.awt.event.KeyListener` interface, method summary, A-14

`java.awt.event.MouseEvent` class, method summary, A-14

`java.awt.event.MouseListener` interface, method summary, A-14

`java.awtFlowLayout` class, method summary, A-12

`java.awt.Font` class, method summary, A-12

`java.awt.Frame` class, method summary, A-12

`java.awt.Graphics` class, method summary, A-12

`java.awt.GridLayout` class, method summary, A-13

`java.awt.Point` class, method summary, A-13

`java.awt.Rectangle` class, method summary, A-13

javadoc comment conventions, 217

javadoc utility, 384–385

`java.io` package, 417, A-14–15

`java.io.EOFException` class, method summary, A-14

`java.io.File` class, method summary, A-14. *See also* `java.util.Scanner` class.

`java.io.FileInputStream` class, method summary, A-14

`java.io.FileNotFoundException`, A-14

`java.io.FileOutputStream` class, method summary, A-14

`java.io.InputStream` class, method summary, A-14–15

`java.io.InputStreamReader` class, method summary, A-15

`java.io.IOException`  
  checked exceptions, 355–356  
  reading web pages, 335  
  summary, A-15  
  throwing an exception, 362–365

`java.io.OutputStream` class, method summary, A-15

`java.io.PrintStream` class, method summary, A-15

`java.io.PrintWriter` class  
  constructing with a string, 335  
  method summary, A-15  
  reading and writing binary data, 336–337  
  writing text files, 332–334

`java.lang` package, 417, A-15–19

`java.lang.AutoCloseable` interface, method summary, A-15

## I-10 Index

- java.lang.Boolean class, method summary, A-15
- java.lang.Character class, method summary, A-15–16
- java.lang.Class class, method summary, A-16
- java.lang.Comparable<T> interface, method summary, A-16
- java.lang.Double class, method summary, A-16–17
- java.lang.Error class, summary, A-16
- java.lang.IllegalArgumentException method summary, A-16  
throwing, 352  
unchecked exceptions, 355
- java.lang.IllegalStateException class, description, A-16
- java.lang.Integer class, method summary, A-16
- java.lang.InterruptedException, A-17
- java.lang.Math class, method summary, A-17–18
- java.lang.NullPointerException class, 360, A-18
- java.lang.NumberFormatException class, 355, A-18
- java.lang.Object class, method summary, A-18
- java.lang.RuntimeException, A-18
- java.lang.String class
  - compareTo method, 92
  - equals method, 92–93
  - length method, 61
  - method summary, A-18–19. *See also specific methods.*
  - substring method, 63
- java.lang.System class, method summary, A-19
- java.lang.Throwable class, method summary, A-19
- java.math package, A-20
- java.math.BigDecimal class, method summary, A-20
- java.math.BigInteger class, method summary, A-20
- java.net package, 417, A-20
- java.net.URL class
  - method summary, A-20
  - reading web pages, 335
- java.sql package, 417t
- java.util package, 417t, A-20–24
- java.util.ArrayList<E> class
  - add method, 302
  - get method, 302
  - method summary, A-20
  - remove method, 304
  - set method, 303
  - size method, 302
- java.util.Arrays class
  - copyOf method, 275–276
  - method summary, A-20–21
  - toString method, 271–272
- java.util.Calendar class, method summary, A-21
- java.util.Collection<E> interface, method summary, A-21
- java.util.Collections class, method summary, A-21
- java.util.Comparator<T> interface, method summary, A-21–22
- java.util.Date class, method summary, A-22
- java.util.EventObject class, method summary, A-22
- java.util.GregorianCalendar class, method summary, A-22
- java.util.HashMap<K, V> class, method summary, A-23
- java.util.HashSet<E> class, method summary, A-22
- java.util.InputMismatchException, A-22
- java.util.Iterator<E> interface, method summary, A-22
- java.util.LinkedList<E> class, method summary, A-22
- java.util.List<E> interface, method summary, A-22
- java.util.ListIterator<E> interface, method summary, A-22–23
- java.util.logging package, A-24
- java.util.logging.Level class, A-24
- java.util.logging.Logger class
  - getGlobal method, 112
  - info method, 112
  - method summary, A-24
  - setLevel method, 112
- java.util.Map<K, V> interface, method summary, A-23
- java.util.NoSuchElementException catching exceptions, 355  
summary, A-23
- java.util.Objects class, method summary, A-23
- java.util.PriorityQueue<E> class, method summary, A-33
- java.util.Queue<E> interface, method summary, A-23
- java.util.Random class, method summary, A-23
- java.util.Scanner class
  - hasNextDouble method, 118
  - hasNextInt method, 118–119
  - method summary, A-24–25
- java.util.Set<E> interface, A-24
- java.util.TreeMap<K, V> class, method summary, A-24
- java.util.TreeSet<E> class, method summary, A-24
- javax.swing package, 417t, A-24–27
- javax.swing.AbstractButton class, method summary, A-24
- javax.swing.ButtonGroup class, method summary, A-24
- javax.swing.event package, A-27
- javax.swing.event.ChangeEvent class, A-27
- javax.swing.event.ChangeListener interface, method summary, A-27
- javax.swing.ImageIcon class, method summary, A-25
- javax.swing.JButton class, method summary, A-25
- javax.swing.JCheckBox class, method summary, A-25
- javax.swing.JComboBox class, method summary, A-25
- javax.swing.JComponent class, method summary, A-25
- javax.swing.JFileChooser class
  - file dialog boxes, 335–336
  - method summary, A-25
- javax.swing.JFrame class, method summary, A-25–26
- javax.swing.JLabel class, method summary, A-26
- javax.swing.JMenuItem class, method summary, A-26
- javax.swing.JMenuBar class, method summary, A-26
- javax.swing.JMenuItem class, method summary, A-26

- `javax.swing.JOptionPane` class, method summary, A-26
- `javax.swing.JPanel` class, constructing, 494–495 for complex frames, 497–498 full code example, 497 grouping user-interface components, 496 method summary, A-26
- `javax.swing.JRadioButton` class, method summary, A-26
- `javax.swing.JScrollPane` class, method summary, A-26
- `javax.swing.JSlider` class, method summary, A-26
- `javax.swing.JTextArea` class, method summary, A-26–27
- `javax.swing.JTextField` class, method summary, A-27
- `javax.swing.KeyStroke` class, method summary, A-27
- `javax.swing.text` package, A-27
- `javax.swing.text.JTextComponent` class, method summary, A-27
- `javax.swing.Timer` class, method summary, A-27
- `JButton` constructor, `javax.swing.JButton` class, A-25
- `JCheckBox` constructor, `javax.swing.JCheckBox` class, A-25
- `JComboBox` constructor, `javax.swing.JComboBox` class, A-25
- `JComponent` class, 515–520, A-25
- `JFileChooser` constructor, `javax.swing.JFileChooser` class, A-25
- `JLabel` constructor, `javax.swing.JLabel` class, A-26
- `JMenu` constructor, `javax.swing.JMenu` class, A-26
- `JMenuBar` constructor, `javax.swing.JMenuBar` class, A-26
- `JMenuItem` constructor, `javax.swing.JMenuItem` class, A-26
- `JPanel`, *See* `javax.swing.JPanel` class
- `JRadioButton` constructor, `javax.swing.JRadioButton` class, A-26
- `JScrollPane` class, `javax.swing.JScrollPane` class, 512–515, A-26
- `JSlider` constructor, `javax.swing.JSlider` class, A-26
- `JTextArea` class, `javax.swing.JTextArea` class, 511–515, A-26–27
- `JTextField` class, `javax.swing.JTextField` class, 509–511, A-27
- justified text (Video Example), 244
- K**
- Kahn, Bob, 470
- keeping a total, patterns for object data, 411
- keyboard input, 50–52, 62
- `keyPressed` method, `java.awt.event.KeyListener` class, A-14
- `keyReleased` method, `java.awt.event.KeyListener` class, A-14
- `keySet` method, `java.util.Map<K, V>` interface, A-23
- `keyTyped` method, `java.awt.event.KeyListener` class, A-14
- L**
- lambda expressions, 479–480, 509
- language support package, 417
- largest values, identifying. *See* `max` method, `java.lang.Math` class.
- `LargestInArray.java` class, 277–278
- `LargestInArrayList.java` class, 309–310
- `length` method, `java.lang.String` class, 61, A-19
- length of strings, computing, 61
- less than or equal ( $\leq$ ), relational operator, 91t
- less than ( $<$ ), relational operator, 91t
- letters. *See* characters; strings.
- lexicographic ordering, 94–95
- library. *See* Java library.
- Licklider, J.C.R., 470
- linear search, 272
- lines (graphic), drawing, 517–520
- lines (of text), reading, 339–340
- listeners. *See* event listeners.
- `listIterator` method, `java.util.List<E>` interface, A-22
- literals in expressions, 43
- loan calculations, Video Example, 399
- local inner classes, 507–508
- local variables, 236
- `log` method, `java.lang.Math` class, 45t, A-17
- `log10` method, `java.lang.Math` class, 46t, A-17
- logging messages, 112
- logic errors, 14
- Loma Prieta earthquake, 98
- loop and a half problem, 162–163
- loops
- asymmetric bounds, 157, 159, 160
  - Boolean variables, 161
  - bounds, choosing, 158
  - break statements, 162–163
  - common errors, 146–148
  - count-controlled, 153
  - counting iterations, 158
  - credit card processing (Worked Example), 174
  - definite, 153
  - definition, 142
  - do loops, 158–159
  - enhanced for loop, 269–270. *See also* `also` for loops.
  - event-controlled, 153
  - flowcharting, 159
  - full code example, 158, 269
  - hand-tracing, 149–152
  - idioms, 157
  - indefinite, 153
  - infinite, 147
  - loop and a half problem, 162–163
  - for loops, 154–158. *See also* enhanced for loops.
  - manipulating pixel images (Worked Example), 177
  - nesting, 174–177
  - off-by-one errors, 147–148
  - post-test, 158
  - pre-test, 158
  - redirecting input/output, 163
  - sentinel values, 160–163
  - symmetric bounds, 157
  - terminating conditions, 146, 157
  - while loops, 142–148, 159
  - writing (How To), 171–174
- loops, common algorithms
- averages, computing, 167
  - comparing adjacent values, 170–171
  - counters, 167–168
  - finding first match, 168
  - full code example, 170
  - maximum/minimum values, finding, 169
  - prompting for first match, 169
  - totals, computing, 167

## I-12 Index

luggage handling system, 97

### M

machine code, 6

magic numbers, 41

main method

  command line arguments, 345–347

  definition, 11

  frame windows, 498

managing object properties, patterns  
  for object data, 413

mathematical methods, 45–46

mathematical operations. *See*  
  arithmetic operations.

matrices. *See* arrays, two-dimensional.

max method, `java.lang.Math` class, 46*t*,  
  A-17

maximum values

  finding in arrays, 271

  finding with loops, 169

maximum values, identifying. *See* max  
  method, `java.lang.Math` class.

MeasurableDemo.java class, 473

Medals.java class, 298–299

menu class, Worked Example, 399

messages. *See* error messages.

methods

  abstract, 456

  accessing data without modifying.  
    *See* accessor methods.

  accessing packages, 421

  accessor, 383

  arguments, 213

  array lists as arguments and  
    return values, 305

  as a black box, 212–213

  body, 214

  comments, 217

  default for interfaces, 477–478

  definition, 11, 212

  dot notation, 66

  duplicate names. *See* overloading.

  examples, 66

  explicit parameters, 388

  final, 457

  full code example, 477

  hand-tracing, 234–235

  headers, 214

  implicit parameters, 388

  instance, 66–67

  invoking on null references, 405

javadoc comment conventions,  
  217

keeping them short, 234

mathematical, 45–46

modifying data. *See* mutator  
  methods.

mutator, 383

with no results. *See* null, testing  
  for.

not invoked on objects, 410

overloading, 393

parameter variables, 214

passing information to. *See*  
  arguments; parameters.

public *versus* private, 388

recursive, 240–242

replacing with stubs, 235–236

results of. *See* return values.

reusable, 225–227

reusing, 393

static, for interfaces, 477

static declaration, 215–216

methods, calling

  definition, 212

  description, 12

  in expressions, 43

  multiply method, 42

  on numbers, 67

  on objects, 66–67

  self-calling. *See* recursive  
    methods.

  superclass methods, 447

methods, implementing

  description, 214–216

  How To, 222–223

methods, static

  definition, 410

  description, 66–67

  full code example, 410

microprocessors, 228

Microsoft product schedules, 111

min method, `java.lang.Math` class, 46*t*,  
  A-17

minimum values

  finding in arrays, 271, 285–286

  finding with loops, 169

  full code example, 285

  identifying. *See* min method, `java.  
lang.Math` class.

misspelling words, 15

mod method, `java.math.BigInteger`  
  class, A-20

modeling moving objects, patterns

  for object data

  description, 414–415

  Video Example, 417

modeling objects with distinct states,  
  413–414

modifying arguments, 219–220

Monte Carlo method, 184–185

MonteCarlo.java, 185

Morris, Robert, 268

Mosaic, 470

mouseClicked method, `java.awt.event.`

`MouseListener` interface, A-14

mouseEntered method, `java.awt.event.`

`MouseListener` interface, A-14

mouseExited method, `java.awt.event.`

`MouseListener` interface, A-14

mousePressed method, `java.awt.event.`

`MouseListener` interface, A-14

mouseReleased method, `java.awt.`

`event.MouseListener` interface,  
    A-14

multidimensional arrays, 301

multiply method

  calling, 42

  description, 42

`java.math.BigDecimal` class, A-20

`java.math.BigInteger` class, A-20

mutator methods, 383

Mycin program, 121

### N

\n newline character, 62

naming conventions

  class names, 35

  constants, 37

  constructors, 389

  packages, 419

  variable names, 35

natural logs, calculating. *See* log

  method, `java.lang.Math` class.

Naughton, Patrick, 6

nesting

  if statements, 102–107

  loops, 174–177

networks, definition, 4

new reserved word, 392

newline character (\n), 62

next method

`java.util.Iterator<E>` interface,  
    A-22

reading strings, 62

next method, `java.util.Scanner` class  
  consuming white space, 337–338, 341  
  description, A-24  
  reading words, 337–338  
nextDouble method, `java.util.Random` class, A-23  
nextDouble method, `java.util.Scanner` class  
  consuming white space, 341  
  description, A-24  
nextInt method, `java.util.Random` class, A-23  
nextInt method, `java.util.Scanner` class  
  consuming white space, 341  
  description, A-24  
  screening integers, 340–341  
nextLine method, `java.util.Scanner` class  
  description, A-24  
  reading lines, 339–340  
Nicely, Thomas, 50  
NoSuchElementException error, 362–365  
null pointer exceptions, 405  
null references  
  definition, 405  
  *versus* empty strings, 405  
  invoking methods on, 405  
  testing for, 405  
number literals, 34t–35  
number sequences, Worked Example, 480  
number types  
  byte data type, 42t  
  char data type, 42t  
  data types, 34–35, 42t  
  double data type, 42t  
  float data type, 42t  
  int data type, 42t  
  long data type, 42t  
  short data type, 42t  
numbering elements of arrays, 264  
numbers. *See also* characters; strings.  
  big, 42  
  converting from strings, 340  
  English words for, 230–234  
  with fractions. *See* floating-point numbers.  
  without fractions. *See* integers.  
integers, 33  
invoking methods on, 67

magic, 41  
primitive types, 65

**O**

Object class. *See also* superclasses.  
  description, 463–464  
  equals method, 465–466  
  instanceof operator, 466–468  
  toString method, 464–465  
  toString method and inheritance, 468–469  
object position, patterns for object data, 414–415  
object references  
  copying, 404  
  definition, 403  
  forgetting to initialize, 407–408  
  null, 405  
  shared, 403–405  
  this, 405–407  
objects  
  of classes, 66–67  
  comparing. *See* equals method.  
  instance variables, initializing. *See* constructors.  
  memory location of. *See* object references.  
  *versus* primitive types, 66–67  
  tracing, 399–402  
OFF message level, 112  
off-by-one errors, 147–148  
OOP (object-oriented)  
  programming). *See also* classes.  
  definition, 376  
  encapsulation, definition, 377  
  overview, 376–378  
open source software, 402  
openStream method, `java.net.URL` class, A-20  
`org.w3c.dom` package, 417t  
out object, `java.lang.System` class, A-19  
output. *See also* input.  
  definition, 4  
  dialog boxes, 67  
  formatting, 52–55  
  redirecting, 163  
ovals, drawing, 517–520  
overloading  
  accidentally, 450  
  methods, 393  
overriding methods  
  preventing, 457

superclass, 443, 446–450, 457  
toString, 464–465

**P**

packages  
  Abstract Window Toolkit, 417t  
  accessing, 421  
  database access, SQL, 417t  
  default, 418  
  definition, 51, 417  
  description, 7  
  disambiguating, 419  
  Document Object Model, 417t  
  import statement, 418–419  
  importing, 51, 418–419  
  input/output, 417  
  Java library, 7  
  language support, 417  
  list of, 417t  
  naming conventions, 419  
  organizing classes into, 417  
  programming with, How To, 421–422  
  source files, 419  
  Swing user interface, 417t  
  utilities, 417t  
paintComponent method, `javax.swing.JComponent` class, 515–524, A-25  
painted components  
  default size, 525  
  drawing, 521–524  
panels, frame windows, 496  
parameter variables  
  definition, 214  
  description, 217–220  
  modifying, 219  
parameters  
  actual, 217–220  
  formal, 217–220  
  passing, 217–220  
  variable number of, 284  
parseDouble method, `java.lang.Double` class  
  converting strings to numbers, 340  
  description, A-16  
parseInt method, `java.lang.Integer` class  
  converting strings to numbers, 340  
  method summary, A-16  
partially filled arrays, 266  
patent, definition, 351

## I-14 Index

- patterns for object data  
  collecting values, 412  
  counting events, 411–412  
  describing object position,  
    414–415  
  keeping a total, 411  
  managing object properties, 413  
  modeling moving objects,  
    description, 414–415, Video  
    Example, 417  
  modeling objects with distinct  
    states, 413–414
- PCs (personal computers),  
  schematic design of, 4. *See also*  
  computers.
- peek method, `java.util.Queue<E>`  
  interface, A-23
- Pentium floating-point bug, 50
- percent sign (%)  
  negative remainders, 49  
  modulus operator, 44  
  in variable names, 35
- peripheral devices, 4
- personal computing, 228
- PGP (Pretty Good Privacy)  
  encryption, 351
- PI constant, `java.lang.Math` class,  
  A-18
- piracy, software, 188
- pixel images, manipulating with  
  loops (Worked Example), 177
- plurals of English words, computing  
  (Video Example), 107
- plus sign (+), concatenation operator,  
  61
- polymorphism. *See also* inheritance.  
  definition, 453  
  dynamic method lookup, 453,  
    455–456  
  implicit parameters, 455–456  
  overview, 452–454
- portability, 7
- post-test loops, 158
- pow method, `java.lang.Math` class, 45*t*,  
  A-17
- powers, calculating. *See* `pow` method,  
  `java.lang.Math` class.
- `PowerTable.java` class, 175–176
- precedence  
  Boolean operators, 113  
  Java operators, A-5
- pre-test loops, 158
- previous method, `java.util.`  
  `ListIterator<E>` interface, A-23
- primary storage, 3
- primitive character types, 65
- primitive number types, 65
- print commands, full code example,  
  13
- print method  
  displaying a prompt, 51  
  `java.io.PrintStream` class, A-15  
  `java.io.PrintWriter` class, 332–  
    334, A-15  
  joining output lines, 13
- printf method  
  `java.io.PrintStream` class, A-15  
  `java.io.PrintWriter` class, 332–  
    334, A-15
- printing  
  arrays, 298–299  
  numerical values, 13  
  return values, 213  
  starting a new line, 13
- println method  
  description, 13  
  `java.io.PrintStream` class, A-15  
  `java.io.PrintWriter` class, 332–  
    334, A-15  
  starting a new line, 13
- printStackTrace method, `java.lang.`  
  `Throwable` class, A-19
- PrintStream constructor  
  `java.io.PrintStream` class, A-15
- PrintWriter constructor  
  `java.io.PrintWriter` class, A-15
- PriorityQueue constructor, `java.util.`  
  `PriorityQueue<E>` class, A-23
- problem solving  
  adapting algorithms, 284–286  
  algorithm design, 15–18  
  discovering algorithms by  
    manipulating physical objects,  
      291–294  
  flowcharts, 107–109  
  patterns for object data, 411–417  
  reusable methods, 225–227  
  selecting test cases, 110–111  
  simpler problems first, 178–182,  
    229–234. *See also* stepwise  
    refinement.
- stepwise refinement, 229–234, 236
- storyboards, 164–166
- tracing objects, 399–402
- problem solving, by hand  
  computing travel time, Worked  
    Example, 60  
  description, 59–60  
  full code example, 59
- program development, examples. *See*  
  “Hello, World” program.
- programming. *See also* applets; Java  
  language.  
  compilers, 6  
  definition, 2  
  getting started. *See* “Hello,  
    World” program.  
  high-level languages, 6  
  machine code, 6  
  scheduling time for, 111–112
- prompting  
  for first match, with loops, 169  
  for input, 51
- protected access feature, 458
- pseudocode  
  for algorithms, 19  
  definition, 17  
  description, 18
- pseudorandom numbers, 183
- public methods, forgetting to declare  
  as public, 475
- put method, `java.util.Map<K, V>`  
  interface, A-23
- pyramid volume, computing  
  (Worked Example), 223
- Q**
- `Question.java` class, 440
- `QuestionDemo1.java` class, 441
- `QuestionDemo2.java` class, 447–448
- `QuestionDemo3.java` class, 454–455
- quotation marks (""), string delimiters  
  definition, 12  
  in string literals, 62
- R**
- \r return character, 63
- radians, converting to degrees. *See*  
  `toDegrees` method, `java.lang.`  
  `Math` class.
- `random` method, `java.lang.Math` class,  
  A-18
- random numbers  
  finding approximate solutions,  
    184–185  
  generating, 182–183

- Monte Carlo method, 185  
 pseudorandom numbers, 183  
 random passwords (Worked Example), 223
- Raymond, Eric, 402
- read method, `java.io.InputStream` class, A-15
- reading exception reports, 286–287
- reading input. *See also* writing output.  
 into arrays, 276–278  
 binary data, 336  
 characters, 338  
 characters from a string, 340  
 classifying characters, 338  
 converting strings to numbers, 340  
 entire files, 344  
 error handling, 361–366  
 lines, 339–340  
 mixed input types, 341  
 from a prompt, 51  
 validating numbers, 340–341  
 web pages (full code example), 335  
 white space, consuming, 337–338  
 words, 337–338
- reading text files  
*How To*, 348–350  
 overview, 332–334
- `readIntBetween` method (full code example), 227
- Rectangle constructor, `java.awt.Rectangle` class, A-13
- rectangles, drawing, 515–517
- recursive methods  
 description, 240–242  
*How To*, 243–244
- redirecting input/output, 163
- regular expressions, 344
- relational operators, 90–91, 115–116
- remainders, calculating. *See* % operator, computing remainders.
- remove method  
`java.util.ArrayList` class, 304, A-20  
`java.util.Collection` interface, A-21  
`java.util.Iterator` interface, A-22  
`java.util.Map` interface, A-23
- `java.util.PriorityQueue` class, A-23
- `removeFirst` method, `java.util.LinkedList` class, A-22
- `removeLast` method, `java.util.LinkedList` class, A-22
- `repaint` method, `java.awt.Component` class, 521–524, A-12
- repainting graphic components  
 forgetting, 524–525  
`repaint` method, 521–524
- `replace` method, `java.lang.String` class, A-19
- `replaceAll` method, `java.lang.String` class, A-19
- reserved words, summary of, A-7t–8t
- return character (\r), 63
- return statement  
 description, 220–221  
 full code example, 221
- return type, void, 224
- return values  
 definition, 213  
 from methods, 213  
 methods without, 224–225  
 missing, 222  
 printing, 213
- reusable methods, 225–227
- reusing algorithms. *See* interface types.
- reusing methods, 393
- `Reverse.java` class, 282–283
- Richter scale, 98t
- Rivest, Ron, 351
- robot escaping from a maze, `Video` Example, 417
- rocket incident, 361
- rolling dice  
 simulating, 183–184  
*Worked Example*, 290
- `round` method, `java.lang.Math` class  
 description, A-18  
 returns, 46t  
 floating-point numbers, 47
- rounding floating-point numbers, 47. *See also* round method.
- roundoff errors, 42, 93–94
- RSA encryption, 351
- run-time errors, 14
- S**
- safety, Java, 7
- sales tax computation, full code example, 401
- Scanner class  
 constructing with a string, 335  
 obtaining Scanner objectives, 51  
 reading and writing binary data, 336–337  
 reading text files, 332–334  
 method summary, `java.util.Scanner` class, A-24
- scheduling time for programming, 111–112
- scope of variables, 236–240
- scroll bars in text input, 512–515
- searching  
 binary, 272, 279–280  
 linear, 272
- secondary storage, 3–4
- semicolon (;)  
 ending statements, 12, 88  
 omitting, 13
- sentinel values, 160–163
- `SentinelDemo.java` class, 160–161
- Sequence interfaces, Worked Example, 480
- sequential search, 272
- set method  
`java.util.ArrayList` class, 303, A-20  
`java.util.ListIterator` interface, A-23
- `setBorder` method, `javax.swing.JComponent` class, A-25
- `setColor` method, `java.awt.Graphics` class, A-12
- `setDefaultCloseOperation` method, `javax.swing.JFrame` class, A-25
- `setEditable` method  
`javax.swing.JComboBox` class, A-25  
`javax.swing.text.JTextComponent` class, A-27
- `setFocusable` method, `java.awt.Component` class, A-12
- `setFont` method, `javax.swing.JComponent` class, A-25
- `setJMenuBar` method, `javax.swing.JFrame` class, A-26
- `setLayout` method, `java.awt.Container` class, A-12

## I-16 Index

- setLevel method, `java.util.logging.Logger` class, 112, A-24
- setLocation method, `java.awt.Rectangle` class, A-13
- setPreferredSize method, `java.awt.Component` class, A-12
- setSelected method, `javax.swing.AbstractButton` class, A-24
- setSelectedItem method, `javax.swing.JComboBox` class, A-25
- setSize method
  - `java.awt.Component` class, A-12
  - `java.awt.Rectangle` class, A-13
- setText method, `javax.swing.text.JTextComponent` class, A-27
- setTitle method, `java.awt.Frame` class, A-12
- setVisible method, `java.awt.Component` class, A-12
- Shamir, Adi, 351
- shared references, 403–405
- shipping costs, computing (full code example), 110
- short circuiting Boolean evaluation, 116–117
- showInputDialog method, `javax.swing.JOptionPane` class, A-26
- showMessageDialog method, `javax.swing.JOptionPane` class, A-26
- showOpenDialog method, `javax.swing.JFileChooser` class, 336, A-25
- showSaveDialog method, `javax.swing.JFileChooser` class, 336, A-25
- simulation programs, 182–186
- sin method, `java.lang.Math` class, 45*t*, A-18
- sine of  $x$ , calculating. *See sin method, java.lang.Math class.*
- size method
  - `java.util.ArrayList<E>` class, 302, A-20
  - `java.util.Collection<E>` interface, A-21
- slash slash (//) comment indicator, 38
- slash asterisk (\*...\*) comment delimiters, 38
- slash asterisks (\*\*...\*\*), comment delimiters, 38, 217
- smallest values, identifying. *See min method, java.lang.Math class.*
- software, definition, 2. *See also programming.*
- software piracy, 188
- sort method
  - `java.util.Arrays` class, A-21
  - `java.util.Collections` class, A-21
- sorting
  - arrays, 279
  - in lexicographic order, 94–95
- source code
  - “Hello, World” program, 11
  - Java programs, 9
- source files, 9
- spaces in
  - comparisons, 95
  - in expressions, 49
  - input. *See white space.*
- spaghetti code, 108–110, 159
- spelling errors, 15
- split method, `java.lang.String` class, A-19
- SQL database access, package for, 417
- sqrt method, `java.lang.Math` class, 45*t*, A-18
- square brackets ([ ]), specifying array slots, 263
- square root, calculating. *See sqrt method, Java.lang.Math class.*
- Stallman, Richard, 402
- start method, `javax.swing.Timer` class, A-27
- stateChanged method, `javax.swing.event.ChangeListener` interface, A-27
- statements
  - definition, 12
  - punctuating, 12–13
- static declaration, methods, 215–216
- static methods
  - definition, 410
  - description, 66–67
  - full code example, 410, 477
  - in interfaces, 477
- static reserved word, 408
- static variables
  - definition, 408–410
  - full code example, 410
- stepwise refinement, 229–234, 236
- stop method, `javax.swing.Timer` class, A-27
- storage devices
  - primary storage, 3
  - secondary storage, 3–4
- storing data in programs. *See constants; variables.*
- storyboards, 164–166. *See also flowcharts.*
- streams, definition, 336
- strings. *See also characters.*
  - versus characters, 63
  - comparing, 94–95
  - concatenating, 61–62
  - converting from objects. *See toString method.*
  - converting to integers. *See parseInt method.*
  - converting to numbers, 340. *See also parseInt method.*
  - counting positions in, 63
  - definition, 12, 61
  - empty, 61
  - escape sequences, 62
  - extracting parts of. *See substrings.*
  - joining. *See concatenating.*
  - length, computing, 61
  - literals, 61
  - reading characters from, 340
  - reading from the console, 62
  - String class. *See java.lang.String class.*
    - Unicode characters, 68
  - stubs, replacing missing methods, 235–236
  - Stuxnet virus, 268
  - subclasses. *See also classes; superclasses.*
    - accidental overloading, 450
    - calling superclass methods, 447
    - constructors, 451
    - declaration, example, 444–445
    - declaration, syntax, 444
    - definition, 438
    - implementing, 442–446
    - inherited methods, 446
    - overriding superclass methods, 446–450
    - replicating instance variables, 445–446
  - substitution principle, 438–439
    - versus superclasses, 446
  - substring method, `java.lang.String` class, A-19

substrings. *See also* strings.  
   extracting, 63–65  
   extracting (Worked Example), 97

subtract method  
   description, 42  
   `java.math.BigDecimal` class, A-20  
   `java.math.BigInteger` class, A-20

summing  
   array elements, 271  
   with loops, 167

super reserved word, 447, 451

superclasses. *See also* classes; `Object` class; subclasses.  
   constructors, 451–452  
   definition, 438  
   overriding superclass methods, 443, 446–450, 457  
   *versus* subclasses, 446  
   substitution principle, 438–439

swapping, array elements, 274

Swing user-interface package, 417t

switch statements, 101. *See also if* statements.

syntax errors, 14

system programmers, 55

**T**

tabs, indenting code, 89

tally counter, 378–381

`tan` method, `java.lang.Math` class, 45t, A-18  
   tangent of  $x$ , calculating. *See tan* method, `java.lang.Math` class.

tax rates, 102t

`TaxCalculator.java` class, 103–104

TCP/IP (Transmission Control Protocol/Internet Protocol), 470

terminating algorithms, 16

test cases, selecting, 110–111

tester classes, 394

testing  
   boundary cases, 110–111  
   classes, 393–399  
   code coverage, 110  
   for `null` references, 405  
   unit tests, 393

text  
   drawing, 517–520  
   justifying (Video Example), 244

text areas, 511–515

text fields, 509–511

text input  
   multiple lines. *See* text areas.  
   scroll bars, 512–515  
   single line. *See* text fields.  
   text areas, 511–515  
   text fields, 509–511

`this` reference, 405–407

throw early, catch late, 359

throw statement, 352–353

`Throwable` constructor, `java.lang.Throwable` class, A-19  
   *throws* clause, 356–357

throwing exceptions  
   full code example, 352  
   overview, 352–353  
   specific exceptions, 360  
   squelching exceptions, 359  
   *throw* early, catch late, 359  
   *try/finally* statements, 360–361

tiling a floor, algorithm for (Worked Example), 21–22

`Timer` constructor, `javax.swing.Timer` class, A-27

`toDegrees` method, `java.lang.Math` class, 45t, A-18

`toLowerCase` method, `java.lang.String` class, A-19

`toRadians` method, `java.lang.Math` class, 45t, A-18

`toString` method  
   full code example, 467  
   inheritance, 468–469  
   `java.lang.Integer` class, A-18  
   `java.lang.Object` class, A-20  
   overriding, 464–465

`toString` method, `java.util.Arrays` class  
   inserting element separators, 271–272  
   summary, A-26

`Total.java` class, 333–334

totals, computing with loops, 167

`toUpperCase` method, `java.lang.String` class, A-19

trace messages, 112

tracing. *See* hand-tracing.

tracing objects, 399–402

transistors, 3

`translate` method, `java.awt.Rectangle` class, A-13

`TreeMap` constructor, `java.util.TreeMap` class, A-24

`TreeSet` constructor, `java.util.TreeSet` class, A-24

`trim` method, 340

trimming white space, 340

try blocks, 354–355

try/catch statement, 354–355

try/finally statement, 360–361

try-with-resources statement, 357

two-dimensional arrays. *See* arrays, two-dimensional.

`TwoRowsOfSquares.java` class, 187–188

types of data. *See* data types.

**U**

unambiguous algorithms, 16

unbalanced parentheses, 48

unchecked exceptions, 355

underscores (`_`), in variable names, 35

unfilled arrays, 267

Unicode, 68, A-1t–3t

uninitialized arrays, 267

unintended integer division, 48

`union` method, `java.awt.Rectangle` class, A-13

`URL` constructor, `java.net.URL` class, A-20

`useDelimiter` method, `java.util.Scanner` class  
   description, A-24  
   reading characters, 338

user-interface components  
   adding to frame windows, 495–496  
   grouping, 496

utilities, package of, 417

**V**

validating numeric input, 340–341

variable names  
   `_` (underscores) in, 35  
   `?` (question mark) in, 35  
   `%` (percent sign) in, 35  
   `$` (dollar sign) in, 35  
   camel case, 35  
   case sensitivity, 35  
   descriptive, 40  
   duplicate, 237–238  
   lowercase letters, 35  
   naming conventions, 35  
   reserved words, 35  
   spaces, in, 35  
   uppercase letters, 35

## I-18 Index

variables. *See also* instance variables; local variables.  
copying, 404  
declaring, 32–34*t*  
defined within methods. *See* local variables.  
definition, 32  
example, 38–39  
in expressions, 43  
final reserved word, 37  
incrementing/decrementing, 43  
local, 236  
scope, 236–240  
syntax, 33  
unchanging. *See* constants.  
undeclared, 39–40  
uninitialized, 39–40  
variables, placing values in initial values. *See* initializing, variables.  
new values. *See* assignment statements.  
variables, static  
definition, 408–410  
full code example, 410

vertical bars (||), *or* operator, 113–114, 116–118  
virtual machine, definition, 7  
viruses, 268  
VisiCalc, 228  
void reserved word  
declaring constructors as, 393  
return type, 224  
volume, computing  
cube, 214–216  
pyramid (Worked Example), 223  
`Volume1.java` class, 38–39  
`Volume2.java` class, 53–55

### W

web pages, reading (full code example), 335  
`while` loops, 142–148, 159. *See also* for loops.  
`while` statement, syntax, 143  
white space  
consuming, 337–338  
trimming, 340  
Wilkes, Maurice, 148

words, reading, 337–338  
world population table (Worked Example), 300  
World Wide Web, 470  
wrapper classes, 305–306, 306*t*  
`write` method, `java.io.OutputStream` class, A-15  
writing output, 342–344. *See also* reading input.  
writing text files  
How To, 348–350  
overview, 332–334

### Z

Zimmerman, Phil, 351

# ILLUSTRATION CREDITS

## Icons

Common Error icon (spider): © John Bell/iStockphoto.  
Computing & Society icon (rhinoceros): © Media Bakery.  
How To icon (compass): © Steve Simzer/iStockphoto.  
Java 8 icon (eightball): © subjug/iStockphoto.  
Paperclips: © Yvan Dube/iStockphoto.  
Programming Tip icon (toucan): Eric Isselé/iStockphoto.  
Self Check icon (stopwatch): © Nicholas Homrich/  
iStockphoto.  
Special Topic icon (tiger): © Eric Isselé/iStockphoto.  
Worked Example icon (globe): © Alex Slobodkin/  
iStockphoto.  
Web only icon (globe): © Alex Slobodkin/iStockphoto.

## Chapter 4

Page 178-181 (left to right, top to bottom):  
Gogh, Vincent van *The Olive Orchard*: Chester Dale  
Collection 1963.10.152/National Gallery of Art.  
Degas, Edgar *The Dance Lesson*: Collection of Mr. and Mrs.  
Paul Mellon 1995.47.6/National Gallery of Art.  
Fragonard, Jean-Honoré *Young Girl Reading*: Gift of Mrs.  
Mellon Bruce in memory of her father, Andrew W.  
Mellon 1961.16.1/National Gallery of Art.  
Gauguin, Paul *Self-Portrait*: Chester Dale Collection  
1963.10.150/National Gallery of Art.  
Gauguin, Paul *Breton Girls Dancing, Pont-Aven*: Collection  
of Mr. and Mrs. Paul Mellon 1983.1.19/National Gallery  
of Art.  
Guigou, Paul *Washerwomen on the Banks of the Durance*:  
Chester Dale Fund 2007.73.1/National Gallery of Art.

Guillaumin, Jean-Baptiste-Armand *The Bridge of Louis  
Philippe*: Chester Dale Collection 1963.10.155/National  
Gallery of Art.  
Manet, Edouard *The Railway*: Gift of Horace Havemeyer  
in memory of his mother, Louise W. Havemeyer  
1956.10.1/National Gallery of Art.  
Manet, Edouard *Masked Ball at the Opera*: Gift of Mrs.  
Horace Havemeyer in memory of her mother-in-law,  
Louise W. Havemeyer 1982.75.1/National Gallery  
of Art.  
Manet, Edouard *The Old Musician*: Chester Dale  
Collection 1963.10.162/National Gallery of Art.  
Monet, Claude *The Japanese Footbridge*: Gift of Victoria  
Nebeker Coberly, in memory of her son John W. Mudd,  
and Walter H. and Leonore Annenberg 1992.9.1/  
National Gallery of Art.  
Monet, Claude *Woman with a Parasol—Madame Monet  
and Her Son*: Collection of Mr. and Mrs. Paul Mellon  
1983.1.29/National Gallery of Art.  
Monet, Claude *The Bridge at Argenteuil*: Collection of Mr.  
and Mrs. Paul Mellon 1983.1.24/National Gallery of Art.  
Monet, Claude *The Artist's Garden in Argenteuil (A Corner  
of the Garden with Dahlias)*: Gift of Janice H. Levin, in  
Honor of the 50th Anniversary of the National Gallery  
of Art 1991.27.1/National Gallery of Art.

## Input

```
Scanner in = new Scanner(System.in);
// Can also use new Scanner(new File("input.txt"));

int n = in.nextInt();
double x = in.nextDouble();
String word = in.next();
String line = in.nextLine();

while (in.hasNextDouble())
{
 double x = in.nextDouble();
 Process x
}
```

## Output

System.out.print("Enter a value: ");

Use + to concatenate values.

System.out.println("Volume: " + volume);

Field width      Precision  
Left-justified string      Integer      Floating-point number

```
System.out.printf("%-10s %10d %.10.2f", name, qty, price);

try (PrintWriter out = new PrintWriter("output.txt"))
{
 Write to out
 Use the print/println/printf methods.
}

The output is closed at the end of
the try-with-resources statement.
```

## Arrays

Element type /  
int[] numbers = new int[5];  
int[] squares = { 0, 1, 4, 9, 16 };  
int[][] magicSquare =  
{  
 { 16, 3, 2, 13 },  
 { 5, 10, 11, 8 },  
 { 9, 6, 7, 12 },  
 { 4, 15, 14, 1 }  
};  
  
for (int i = 0; i < numbers.length; i++)
{
 numbers[i] = i \* i;
}  
  
for (int element : numbers)
{
 Process element
}  
  
System.out.println(Arrays.toString(numbers));
// Prints [0, 1, 4, 9, 16]

## Array Lists

Use wrapper type, Integer, Double, etc., for primitive types.  
Initially empty  
Element type (optional)  
ArrayList<String> names = new ArrayList<String>();  
Add elements to the end  
names.add("Ann");  
names.add("Cindy"); // [Ann, Cindy], names.size() is now 2  
names.add(1, "Bob"); // [Ann, Bob, Cindy]  
names.remove(2); // [Ann, Bob]  
names.set(1, "Bill"); // [Ann, Bill]  
  
String name = names.get(0); // Gets "Ann"  
System.out.println(names); // Prints [Ann, Bill]

## Linked Lists, Sets, and Iterators

LinkedList<String> names = new LinkedList<>();
names.add("Bob"); // Adds at end  
  
ListIterator<String> iter = names.listIterator();
iter.add("Ann"); // Adds before current position  
  
String name = iter.next(); // Returns "Ann"
iter.remove(); // Removes "Ann"  
  
Set<String> names = new HashSet<>();
names.add("Ann"); // Adds to set if not present
names.remove("Bob"); // Removes if present  
  
Iterator<String> iter = names.iterator();
while (iter.hasNext())
{
 Process iter.next()
}

## Maps

Key type / Value type /  
Map<String, Integer> scores = new HashMap<>();  
scores.put("Bob", 10);  
Integer score = scores.get("Bob");  
Returns null if key not present  
  
for (String key : scores.keySet())
{
 Process key and scores.get(key)
}

# **WILEY END USER LICENSE AGREEMENT**

Go to [www.wiley.com/go/eula](http://www.wiley.com/go/eula) to access Wiley's ebook EULA.